Learn Java/J2EE core concepts and design/coding issues

With

Java/J2EE Job Interview Companion

By

K.Arulkumaran

Technical Reviewers

Craig Malone
Lara D'Albreo
Stuart Watson

Acknowledgements

A. Sivayini
R.Kumaraswamipillai

Cover Design

K. Arulkumaran
A.Sivayini

Java/J2EE
Job Interview Companion

Copy Right 2005 K.Arulkumaran

The author has made every effort in the preparation of this book to ensure the accuracy of the information. However, information in this book is sold without warranty either express or implied. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Outline

SECTION	DESCRIPTION
	What this book will do for you?
	Motivation for this book
	Key Areas index
SECTION 1	Interview questions and answers on: **Java** ▪ Language Fundamentals ▪ Swing ▪ Applet ▪ Performance and memory Leaks. ▪ Personal
SECTION 2	Interview questions and answers on: **Enterprise Java** ▪ J2EE ▪ Servlet ▪ JSP ▪ JDBC ▪ JNDI ▪ RMI ▪ EJB ▪ JMS ▪ XML ▪ SQL, Database tuning and O/R mapping ▪ RUP & UML ▪ Struts ▪ Web and Application servers. ▪ Best practices and performance considerations. ▪ Testing and deployment. ▪ Personal
SECTION 3	Putting it all together section. **How would you go about…?** 1. How would you go about documenting your Java/J2EE application? 2. How would you go about designing a Java/J2EE application? 3. How would you go about identifying performance problems and/or memory leaks in your Java application? 4. How would you go about minimising memory leaks in your Java/J2EE application? 5. How would you go about improving performance of your Java/J2EE application? 6. How would you go about identifying any potential thread-safety issues in your Java/J2EE application? 7. How would you go about identifying any potential transactional issues in your Java/J2EE application? 8. How would you go about applying the Object Oriented (OO) design concepts in your Java/J2EE

	application? 9. How would you go about applying the UML diagrams in your Java/J2EE project? 10. How would you go about describing the software development processes you are familiar with? 11. How would you go about applying the design patterns in your Java/J2EE application? 12. How would you go about determining the enterprise security requirements for your Java/J2EE application? 13. How would you go about describing the open source projects like JUnit (unit testing), Ant (build tool), CVS (version control system) and log4J (logging tool) which are integral part of most Java/J2EE projects? 14. How would you go about describing Web services?
SECTION 4	**Emerging Technologies/Frameworks** ▪ Test Driven Development (**TDD**). ▪ Aspect Oriented Programming (**AOP**). ▪ Inversion of Control (**IOC**) (Also known as **Dependency Injection**). ▪ Annotations or attributes based programming (xdoclet etc). ▪ Spring framework. ▪ Hibernate framework. ▪ EJB 3.0. ▪ JavaServer Faces (**JSF**) framework.
SECTION 5	**Sample interview questions …** ▪ **Java** ▪ **Web Components** ▪ **Enterprise** ▪ **Design** ▪ **General**
	GLOSSARY OF TERMS
	RESOURCES
	INDEX

Table of contents

Outline 3
Table of contents 5
What this book will do for you? 7
Motivation for this book 8
Key Areas Index 10
Java – Interview questions & answers 11
- Java – Language Fundamentals 12
- Java – Swing 44
- Java – Applet 48
- Java – Performance and Memory leaks 50
- Java – Personal 53
- Java – Key Points 56

Enterprise Java – Interview questions & answers 59
- Enterprise - J2EE 60
- Enterprise - Servlet 69
- Enterprise - JSP 77
- Enterprise - JDBC 83
- Enterprise – JNDI & LDAP 87
- Enterprise - RMI 90
- Enterprise – EJB 2.x 94
- Enterprise - JMS 110
- Enterprise - XML 114
- Enterprise – SQL, Tuning and O/R mapping 119
- Enterprise - RUP & UML 126
- Enterprise - Struts 133
- Enterprise - Web and Application servers 137
- Enterprise - Best practices and performance considerations 139
- Enterprise – Logging, testing and deployment 141
- Enterprise - Personal 144
- Enterprise – Software development process 144
- Enterprise – Key Points 146

How would you go about…? 151
- Q 01: How would you go about documenting your Java/J2EE application? 152
- Q 02: How would you go about designing a Java/J2EE application? 153
- Q 03: How would you go about identifying performance and/or memory issues in your Java/J2EE application? 156
- Q 04: How would you go about minimising memory leaks in your Java/J2EE application? 157
- Q 05: How would you go about improving performance in your Java/J2EE application? 157
- Q 06: How would you go about identifying any potential thread-safety issues in your Java/J2EE application? 158
- Q 07: How would you go about identifying any potential transactional issues in your Java/J2EE application? 159
- Q 08: How would you go about applying the Object Oriented (OO) design concepts in your Java/J2EE application? 160
- Q 09: How would you go about applying the UML diagrams in your Java/J2EE project? 162

Q 10: How would you go about describing the software development processes you are familiar with? 163

Q 11: How would you go about applying the design patterns in your Java/J2EE application? 165

Q 12: How would you go about determining the enterprise security requirements for yor Java/J2EE application? 194

Q 13: How would you go about describing the open source projects like JUnit (unit testing), Ant (build tool), CVS (version control system) and log4J (logging tool) which are integral part of most Java/J2EE projects? 199

Q 14: How would you go about describing Web services? 206

Emerging Technologies/Frameworks… 210

Q 01: What is Test Driven Development (TDD)? 211

Q 02: What is the point of Test Driven Development (TDD)? 211

Q 03: What is aspect oriented programming? Explain AOP? 212

Q 04: What are the differences between OOP and AOP? 214

Q 05: What are the benefits of AOP? 214

Q 06: What is attribute or annotation oriented programming? 215

Q 07: What are the pros and cons of annotations over XML based deployment descriptors? 215

Q 08: What is XDoclet? 216

Q 09: What is inversion of control (IOC) (also known as dependency injection)? 216

Q 10: What are the different types of dependency injections? 217

Q 11: What are the benefits of IOC (aka Dependency Injection)? 217

Q 12: What is the difference between a service locator pattern and an inversion of control pattern? 217

Q 13: Why dependency injection is more elegant than a JNDI lookup to decouple client and the service? 218

Q 14: Explain Object-to-Relational (O/R) mapping? 218

Q 15: Give an overview of hibernate framework? 218

Q 16: Explain some of the pitfalls of Hibernate and explain how to avoid them? 220

Q 17: Give an overview of the Spring framework? 221

Q 18: How would EJB 3.0 simplify your Java development compared to EJB 1.x, 2.x? 222

Q 19: Briefly explain key features of the JavaServer Faces (JSF) framework? 223

Q 20: How would the JSF framework compare with the Struts framework? 225

Sample interview questions… 226

Java 227

Web components 227

Enterprise 227

Design 229

General 229

GLOSSARY OF TERMS 230

RESOURCES 232

INDEX 234

What this book will do for you?

Have you got the time to read 10 or more books and articles to add value prior to the interview? This book has been written mainly from the perspective of **Java/J2EE job seekers** and **interviewers**. There are numerous books and articles on the market covering specific topics like Java, J2EE, EJB, Design Patterns, ANT, CVS, Multi-Threading, Servlets, JSP, emerging technologies like AOP (Aspect Oriented Programming), Test Driven Development (TDD), Inversion of Control (IoC) etc. But from an interview perspective it is not possible to brush up on all these books where each book usually has from 300 pages to 600 pages. The basic purpose of this book is to cover all the core concepts and design/coding issues which, all Java/J2EE developers, designers and architects should be conversant with to perform well in their current jobs and to launch a successful career by doing well at interviews. The interviewer can also use this book to make sure that they hire the right candidate depending on their requirements. This book contains a wide range of topics relating to Java/J2EE development in a concise manner supplemented with diagrams, tables, sample codes and examples. This book is also appropriately categorised to enable you to choose the area of interest to you.

This book will assist all Java/J2EE practitioners to become better at what they do. Usually it takes years to understand all the core concepts and design/coding issues when you rely only on your work experience. The best way to fast track this is to read appropriate technical information and proactively apply these in your work environment. It worked for me and hopefully it will work for you as well. I was also at one stage undecided whether to name this book "**Java/J2EE core concepts and solving design/coding issues**" or "**Java/J2EE Job Interview Companion**". The reason I chose "**Java/J2EE Job Interview Companion**" is because these core concepts and design/coding issues helped me to be successful in my interviews and also gave me thumbs up in code reviews.

Motivation for this book

I started using Java in 1999 when I was working as a junior developer. During those two years as a permanent employee, I pro-actively spent many hours studying the core concepts behind Java/J2EE in addition to my hands on practical experience. Two years later I decided to start contracting. Since I started contracting in 2001, my career had a much-needed boost in terms of contract rates, job satisfaction, responsibility etc. I moved from one contract to another with a view of expanding my skills and increasing my contract rates.

In the last 5 years of contracting, I have worked for 5 different organisations both medium and large on 8 different projects. For each contract I held, on average I attended 6-8 interviews with different companies. In most cases multiple job offers were made and consequently I was in a position to negotiate my contract rates and also to choose the job I liked based on the type of project, type of organisation, technology used, etc. I have also sat for around 10 technical tests and a few preliminary phone interviews.

The success in the interviews did not come easily. I spent hours prior to each set of interviews wading through various books and articles as a preparation. The motivation for this book was to collate all this information into a single book, which will save me time prior to my interviews but also can benefit others in their interviews. What is in this book has helped me to go from **just a Java/J2EE job to a career in Java/J2EE** in a short time. It has also given me the job security that 'I can find a contract/permanent job opportunity even in the difficult job market'.

I am not suggesting that every one should go contracting but by performing well at the interviews you can be in a position to pick the permanent role you like and also be able to negotiate your salary package. Those of you who are already in good jobs can impress your team leaders, solution designers and/or architects for a possible promotion by demonstrating your understanding of the key areas discussed in this book. You can discuss with your senior team members about **performance issues, transactional issues, threading issues (concurrency issues)** and **memory issues**. In most of my previous contracts I was in a position to impress my team leads and architects by pinpointing some of the critical performance, memory, transactional and threading issues with the code and subsequently fixing them. Trust me it is not hard to impress someone if you understand the key areas.

For example:

- Struts action classes are not thread-safe (Refer **Q113** in Enterprise section).
- JSP variable declaration is not thread-safe (Refer **Q34** in Enterprise section).
- Valuable resources like database connections should be closed properly to avoid any memory and performance issues (Refer **Q45** in Enterprise section).
- Throwing an application exception will not rollback the transaction in EJB. (Refer **Q77** in Enterprise section).

The other key areas, which are vital to any software development, are a good understanding of some of **key design concepts, design patterns,** and a **modelling language** like **UML**. These key areas are really worthy of a mention in your resume and interviews.

For example:

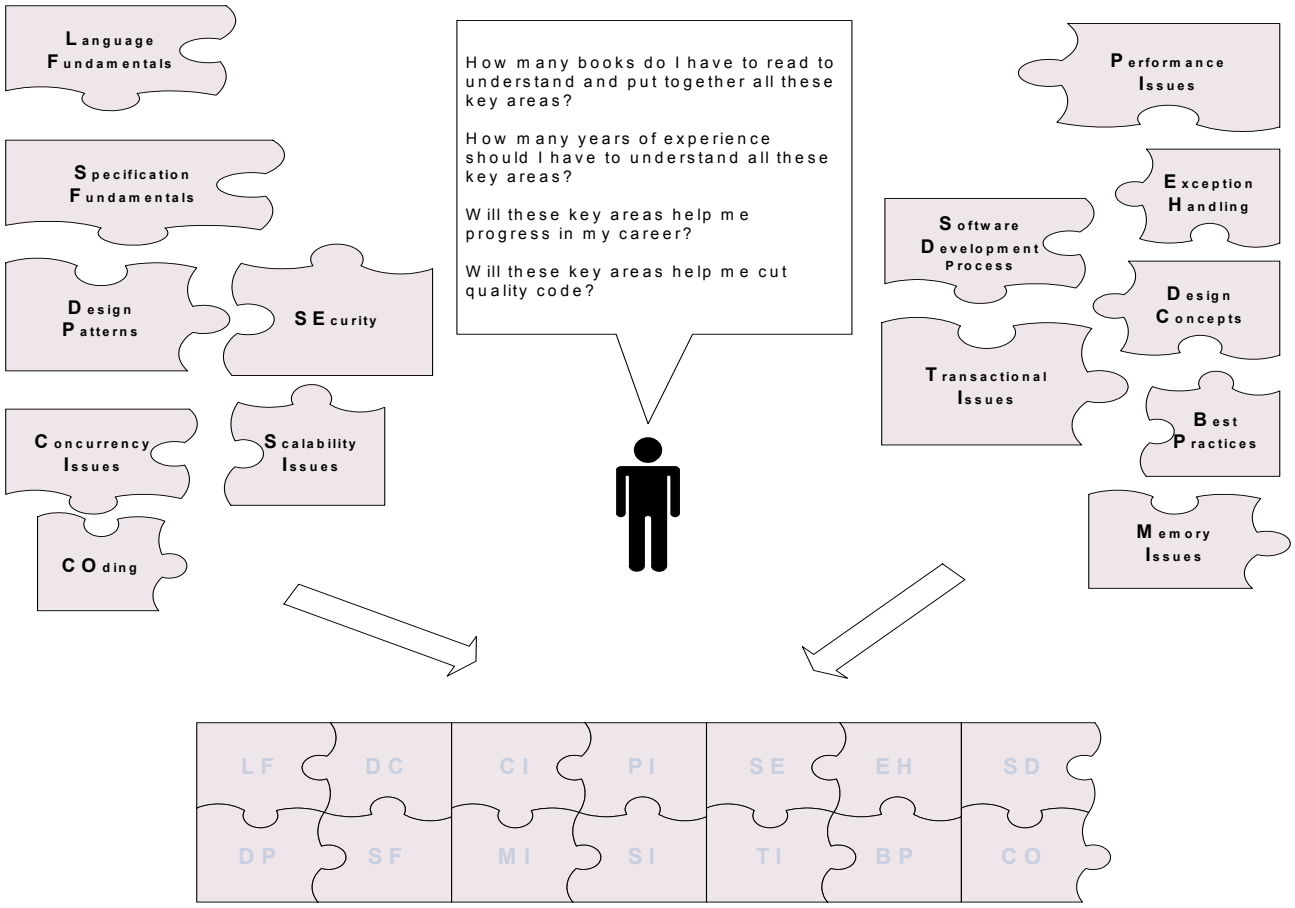
- Know how to use inheritance, polymorphism and encapsulation (Refer **Q5**, **Q6**, **Q7**, and **Q8** in Java section.).
- Why use design patterns? (Refer **Q5** in Enterprise section).
- Why is UML important? (Refer **Q106** in Enterprise section).

If you happen to be in an interview with an organization facing serious issues with regards to their Java application relating to memory leaks, performance problems or a crashing JVM etc then you are likely to be asked questions on these topics. Refer **Q 63 – Q 65** in Java section and **Q123, Q125** in Enterprise section.

Another good reason why these key areas like transactional issues, design concepts, design patterns etc are vital are because solution designers, architects, team leads, and/or senior developers are usually responsible for conducting the technical interviews. These areas are their favourite topics because these are essential to any software development.

Some interviewers request you to write a small program during interview or prior to getting to the interview stage. This is to ascertain that you can code using object oriented concepts and design patterns. So I have included a **coding key area** to illustrate what you need to look for while coding.

- Apply OO concepts like inheritance, polymorphism and encapsulation: Refer **Q08** in Java section.
- Program to interfaces not to implementations: Refer **Q08, Q15** in Java section.
- Use of relevant design patterns: Refer **Q11** in How would you go about… section.
- Use of Java collection API and exceptions correctly: Refer **Q15, Q34**, and **Q35** in Java section.
- Stay away from hard coding values: Refer **Q04** in Java section.



This book aims to solve the above dilemma.

My dad keeps telling me to find a permanent job (instead of contracting), which in his view provides better job security but I keep telling him that in my view in Information Technology the job security is achieved only by keeping your knowledge and skills sharp and up to date. The 8 contract positions I held over the last 5.5 years have given me broader experience in Java/J2EE and related technologies. It also kept me motivated since there was always something new to learn in each assignment, and not all companies will appreciate your skills and expertise until you decide to leave. Do the following statements sound familiar to you when you hand in your resignation or decide not to extend your contract after getting another job offer? "Can I tempt you to come back? What can I do to keep you here?" etc. You might even think why you waited so long. The best way to make an impression in any organisations is to understand and proactively apply and resolve the issues relating to the **Key Areas** discussed in the next section. But **be a team player, be tactful** and **don't be critical of everything, do not act in a superior way** and **have a sense of humour.**

"Technical skills must be complemented with interpersonal skills."

Quick Read guide: It is recommended that you go through all the questions in all the sections but if you are pressed for time or would like to read it just before an interview then follow the steps shown below:

1. Read/Browse **Popular Questions** in Java and Enterprise Java sections.
2. Read/Browse **Key Points** in Java and Enterprise Java sections.
3. Read/Browse through "**Emerging Technologies/Frameworks**" section.
4. Read/Browse "**How would you go about…**" section excluding **Q11** & **Q13**, which are discussed in detail**.**

Key Areas Index

I have categorised the core concepts and issues into **14 key areas** as listed below. These key areas are vital for any good software development. This index will enable you to refer to the questions based on **key areas**. Also note that each question has an icon next to it to indicate which key area or areas it belongs to. Additional reading is recommended for beginners in each of the key areas.

Key Areas	icon	Question Numbers			
		Java section	Enterprise section	How would you go about…?	Emerging Technologies /Frameworks
Language Fundamentals	LF	Q1-Q4, Q10-Q14, Q16-Q20, Q22-Q27, Q30-Q33, Q36-Q43, Q47-Q62	-		Q10, Q15, Q17, Q19
Specification Fundamentals	SF	-	Q1-Q19, Q26-Q33, Q35-Q38, Q41, Q42, Q44, Q46-Q81, Q89-Q97, Q99, 102, Q110, Q112-Q115, Q118-Q119, Q121, Q126, Q127, Q128	Q14	
Design Concepts	DC	Q5-Q9, Q10, Q13, Q22, Q49	Q2, Q3, Q19, Q20, Q21, Q31, Q45, Q98, Q106, Q107, Q108, Q109, 101, Q111	Q02, Q08, Q09	Q3-Q9, Q11, Q13, Q14, Q16, Q18, Q20
Design Patterns	DP	Q10, Q14, Q20, Q31, Q45, Q46, Q50, Q54, Q66	Q5, Q5, Q22, Q24, Q25, Q83, Q84, Q85, Q86, Q87, Q88, Q110, Q111, Q116	Q11	Q12
Transactional Issues	TI	-	Q43, Q71, Q72, Q73, Q74, Q75, Q77, Q78, Q79	Q7	
Concurrency Issues	CI	Q13, Q15, Q29, Q36, Q40, Q53	Q16, Q34, Q113	Q6	
Performance Issues	PI	Q13, Q15 -Q22, Q40, Q53, Q63.	Q10, Q16, Q43, Q45, Q46, Q72, Q83-Q88, Q97, Q98, Q100, Q102, Q123, Q125, Q128	Q3, Q5	
Memory Issues	MI	Q22, Q29, Q32, Q33, Q36, Q45, Q64, Q65.	Q45, Q93	Q3, Q4	
Scalability Issues	SI	Q19, Q20	Q20, Q21, Q120, Q122		
Exception Handling	EH	Q34,Q35	Q76, Q77		
Security	SE	Q61	Q12, Q13, Q23, Q35, Q46, Q51, Q58, Q81	Q12	
Best Practices	BP	Q15, Q21, Q34, Q63, Q64	Q10, Q16, Q39, Q40, Q46, Q82, Q124, Q125		
Software Development Process	SD	-	Q103-Q109, Q129, Q133, Q134, Q136	Q1, Q10, Q13	Q1, Q2
Coding[1]	CO	Q04, Q08, Q10, Q12, Q13, Q15, Q16, Q17, Q21, Q34, Q45, Q46	Q10, Q18, Q21, Q23, Q36, Q38, Q42, Q43, Q45, Q74, Q75, Q76, Q77, Q112, Q114, Q127, Q128	Q11	

[1] Some interviewers request you to write a small program during interview or prior to getting to the interview stage. This is to ascertain that you can code using object oriented concepts and design patterns. I have included a coding key area to illustrate what you need to look for while coding. Unlike other key areas, the CO is not always shown against the question but shown above the actual section of relevance within a question.

SECTION ONE

Java – Interview questions & answers

KEY AREAS

- Language Fundamentals LF
- Design Concepts DC
- Design Patterns DP
- Concurrency Issues CI
- Performance Issues PI
- Memory Issues MI
- Exception Handling EH
- Security SE
- Scalability Issues SI
- Coding[1] CO

Popular Questions: Q01, Q04, Q07, Q08, Q13, Q16, Q17, Q18, Q19, Q25, Q27, Q29, Q32, Q34, Q40, Q45, Q46, Q50, Q51, Q53, Q54, Q55, Q63,Q64, Q66, **Q67**

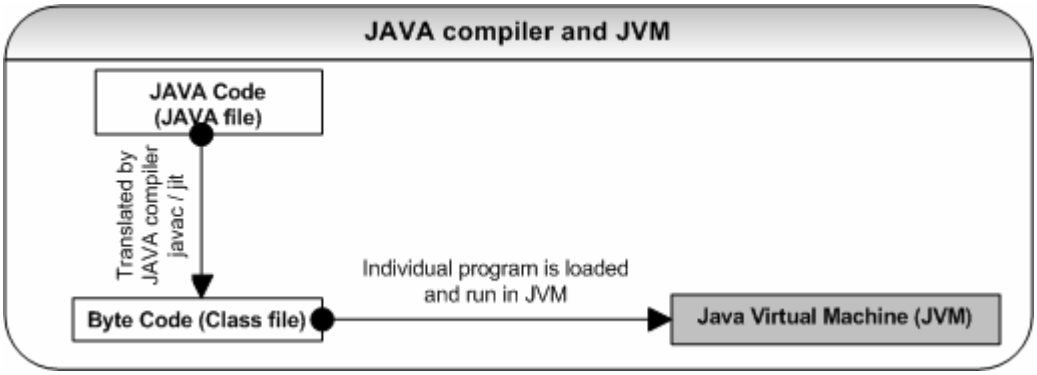
[1] Unlike other key areas, the CO is not always shown against the question but shown above the actual subsection of relevance within a question.

Java – Language Fundamentals

Q 01: Give a few reasons for using Java? LF DC
A 01: Java is a fun language. Let's look at some of the reasons:

- Built-in support for multi-threading, socket communication, and memory management (automatic garbage collection).
- Object Oriented (OO).
- Better portability than other languages across operating systems.
- Supports Web based applications (Applet, Servlet, and JSP), distributed applications (sockets, RMI. EJB etc) and network protocols (HTTP, JRMP etc) with the help of extensive standardised APIs (Application Program Interfaces).

Q 02: What is the main difference between the Java platform and the other software platforms? LF
A 02: Java platform is a software-only platform, which runs on top of other hardware-based platforms like UNIX, NT etc.



The Java platform has 2 components:

- Java Virtual Machine (**JVM**) – 'JVM' is a software that can be ported onto various hardware platforms. Byte codes are the machine language of the JVM.
- Java Application Programming Interface (**Java API**) -

Q 03: What is the difference between C^{++} and Java? LF
A 03: Both C++ and Java use similar syntax and are Object Oriented, but:

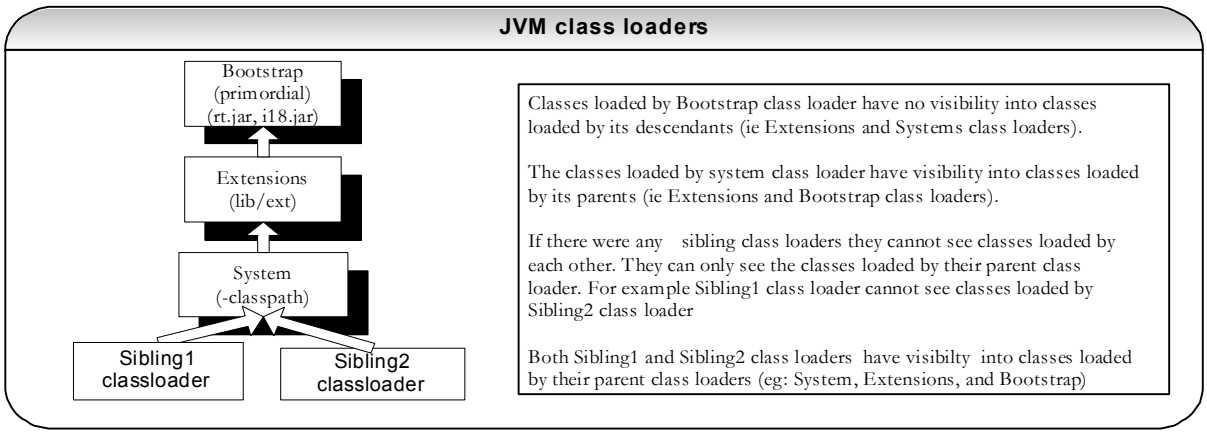
- Java does not support pointers. Pointers are inherently tricky to use and troublesome.
- Java does not support multiple inheritances because it causes more problems than it solves. Instead Java supports **multiple interface inheritance**, which allows an object to inherit many method signatures from different interfaces with the condition that the inheriting object must implement those inherited methods. The multiple interface inheritance also allows an object to behave **polymorphically** on those methods. [Refer **Q 8** and **Q 10** in Java section.]
- Java does not support destructors but rather adds a finalize() method. Finalize methods are invoked by the garbage collector prior to reclaiming the memory occupied by the object, which has the finalize() method. This means you do not know when the objects are going to be finalized. **Avoid using finalize() method to release non-memory resources** like file handles, sockets, database connections etc because Java has only a finite number of these resources and you do not know when the garbage collection is going to kick in to release these resources through the finalize() method.
- Java does not include structures or unions because the traditional data structures are implemented as an object oriented framework (Java collection framework – Refer **Q14**, **Q15** in Java section).

- All the code in Java program is encapsulated within classes therefore Java does not have global variables or functions.
- C++ requires explicit memory management, while Java includes automatic garbage collection. [Refer **Q32** in Java section].

Q 04: Explain Java class loaders? Explain dynamic class loading? LF

A 04: Class loaders are hierarchical. Classes are introduced into the JVM as they are referenced by name in a class that is **already running** in the JVM. So how is the very first class loaded? The very first class is specially loaded with the help of *static main()* method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded and running. A class loader creates a namespace. All JVMs include at least one class loader that is embedded within the JVM called the primordial (or bootstrap) class loader. Now let's look at non-primordial class loaders. The JVM has hooks in it to allow user defined class loaders to be used in place of primordial class loader. Let us look at the class loaders created by the JVM.

CLASS LOADER	reloadable?	Explanation
Bootstrap (**primordial**)	No	Loads JDK internal classes, *java.** packages. (as defined in the sun.boot.class.path system property, typically loads rt.jar and i18n.jar)
Extensions	No	Loads jar files from JDK extensions directory (as defined in the java.ext.dirs system property – usually lib/ext directory of the JRE)
System	No	Loads classes from system classpath (as defined by the java.class.path property, which is set by the **CLASSPATH** environment variable or –classpath or –cp command line options)



Class loaders are hierarchical and use a **delegation model** when loading a class. Class loaders request their parent to load the class first before attempting to load it themselves. When a class loader loads a class, the child class loaders in the hierarchy will never reload the class again. Hence **uniqueness** is maintained. Classes loaded by a child class loader have **visibility** into classes loaded by its parents up the hierarchy but the reverse is not true as explained in the above diagram.

Important: Two objects loaded by different class loaders are never equal even if they carry the same values, which mean a class is uniquely identified in the context of the associated class loader. This applies to **singletons** too, where **each class loader will have its own singleton**. [Refer **Q45** in Java section for singleton design pattern]

Explain static vs. dynamic class loading?

Static class loading	Dynamic class loading

Classes are statically loaded with Java's "new" operator.
 	
`class MyClass {`
`  public static void main(String args[]) {`
`  Car c = new Car();`
`  }`
`}`
	Dynamic loading is a technique for programmatically invoking the functions of a class loader at run time. Let us look at how to load classes dynamically.
	
	Class.forName (String *className*); //static method which returns a Class
	
	The above static method returns the class object associated with the class name. The string *className* can be supplied dynamically at run time. Unlike the static loading, the dynamic loading will decide whether to load the class *Car* or the class *Jeep at* runtime based on a properties file and/or other runtime

	conditions. Once the class is dynamically loaded the following method returns an instance of the loaded class. It's just like creating a class object with no arguments.
	
	class.newInstance (); //A non-static method, which creates an instance of a class (i.e. creates an object).
	
	Jeep myJeep = null ;
	//myClassName should be read from a properties file or Constants interface. //stay away from hard coding values in your program. **CO**
	String myClassName = "au.com.Jeep" ;
	Class vehicleClass = **Class.forName**(myClassName) ;
	myJeep = (Jeep) vehicleClass.**newInstance**();
	myJeep.setFuelCapacity(50);

A ***NoClassDefFoundException*** is thrown if a class is referenced with Java's "*new"* operator (i.e. static loading) but the runtime system cannot find the referenced class.	A ***ClassNotFoundException*** is thrown when an application tries to load in a class through its string name using the following methods but no definition for the class with the specified name could be found: ▪ The forName(..) method in class - ***Class***. ▪ The findSystemClass(..) method in class - ***ClassLoader***. ▪ The loadClass(..) method in class - ***ClassLoader***.

What are "static initializers" or "static blocks with no function names"? When a class is loaded, all blocks that are declared static and don't have function name (i.e. static initializers) are executed even before the constructors are executed. As the name suggests they are typically used to initialize static fields. **CO**

```
public class StaticInitilaizer {
  public static final int A = 5;
  public static final int B;

  //Static initializer block, which is executed only once when the class is loaded.

  static {
    if(A == 5)
      B = 10;
    else
      B = 5;
  }

  public StaticInitilaizer(){}  // constructor is called only after static initializer block
}
```

The following code gives an **Output of** A=5, B=10.

```
public class Test {
   System.out.println("A =" + StaticInitilaizer.A + ", B =" + StaticInitilaizer.B);
}
```

Q 05: What are the advantages of Object Oriented Programming Languages (OOPL)? **DC**

A 05: The Object Oriented Programming Languages directly represent the real life objects like *Car*, *Jeep*, *Account*, *Customer* etc. The features of the OO programming languages like ***polymorphism, inheritance*** *and* ***encapsulation*** make it powerful. [**Tip:** remember **pie** which, stands for **P**olymorphism**, I**nheritance and **E**ncapsulation are the **3 pillars** of OOPL]

Q 06: How does the Object Oriented approach improve software development? **DC**

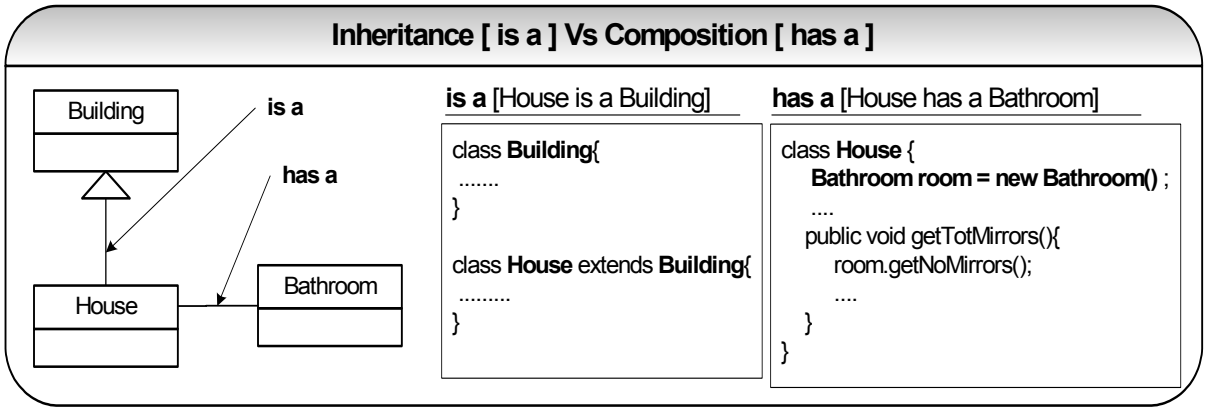
A 06: The key benefits are:

- ***Re-use*** of previous work: using **implementation inheritance** and **object composition**.
- ***Real mapping to the problem domain:*** Objects map to real world and represent vehicles, customers, products etc: with **encapsulation**.
- ***Modular Architecture:*** Objects, systems, frameworks etc are the building blocks of larger systems.

The ***increased quality*** and ***reduced development time*** are the by-products of the key benefits discussed above. If 90% of the new application consists of proven existing components then only the remaining 10% of the code have to be tested from scratch.

Q 07: How do you express an *'**is a**'* relationship and a *'**has a**'* relationship or explain inheritance and composition? What is the difference between composition and aggregation? **DC**

A 07: The *'**is a**'* relationship is expressed with **inheritance** and *'**has a**'* relationship is expressed with **composition**. Both inheritance and composition allow you to place sub-objects inside your new class. Two of the main techniques for **code reuse** are **class inheritance** and **object composition.**



Inheritance is uni-directional. For example *House* **is a** *Building*. But *Building* is not a *House*. Inheritance uses **extends** key word. **Composition*:*** is used when *House* **has a** *Bathroom.* It is incorrect to say *House* is a *Bathroom*. Composition simply means using instance variables that refer to other objects. The class *House* will have an instance variable, which refers to a *Bathroom* object.

Which one to use? The guide is that inheritance should be only used when *subclass* '**is a**' *superclass*.

- Don't use inheritance just to get code reuse. If there is no '**is a**' relationship then use composition for code reuse. Overuse of **implementation inheritance** (uses the "extends" key word) can break all the subclasses, if the superclass is modified.

- Do not use inheritance just to get polymorphism. If there is no '**is a**' relationship and all you want is **polymorphism** then use **interface inheritance** with **composition,** which gives you **code reuse** (Refer **Q8** in Java section for interface inheritance).

What is the difference between aggregation and composition?

Aggregation	Composition
Aggregation is an association in which one class belongs to a collection. This is a part of a whole relationship where a part can exist without a whole. **For example** a line item is a whole and product is a part. If a line item is deleted then corresponding product need not be deleted. So **aggregation has a weaker relationship**.	Composition is an association in which one class belongs to a collection. This is a part of a whole relationship where a part cannot exist without a whole. If a whole is deleted then all parts are deleted. **For example** An order is a whole and line items are parts. If an order deleted then all corresponding line items for that order should be deleted. So **composition has a stronger relationship**.

Q 08: What do you mean by polymorphism, inheritance, encapsulation, and dynamic binding? **DC**

A 08: ***Polymorphism*** – means the ability of a single variable of a given type to be used to reference objects of different types, and automatically call the method that is specific to the type of object the variable references. In a nutshell, polymorphism is a bottom-up method call. The benefit of polymorphism is that it is **very easy to add new classes of derived objects without breaking the calling code** (i.e. getTotArea() in the sample code shown below) that uses the polymorphic classes or interfaces. When you send a message to an object even though you don't know what specific type it is, and the right thing happens, that's called ***polymorphism***. The process used by object-oriented programming languages to implement polymorphism is called ***dynamic binding***. Let us look at some sample code to demonstrate polymorphism: **CO**

Sample code:

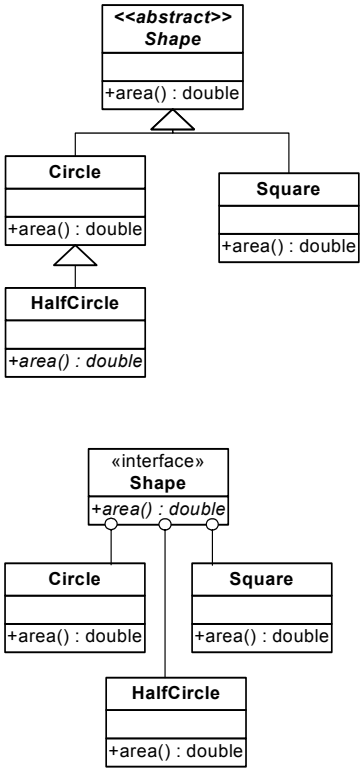
```
//client or calling code
double dim = 5.0; //ie 5 meters radius or width
List listShapes = new ArrayList(20);

Shape s = new Circle();
 listShapes.add(s); //add circle

s = new Square();
listShapes.add(s); //add square

getTotArea (listShapes,dim); //returns 78.5+25.0=103.5

//Later on, if you decide to add a half circle then define
//a HalfCircle  class, which extends Circle and then provide an
//area(). method but your called method getTotArea(...) remains
//same.

s = new HalfCircle();
listShapes.add(s); //add HalfCircle

getTotArea (listShapes,dim); //returns 78.5+25.0+39.25=142.75

/** called method: method which adds up areas of various
**  shapes supplied to it.
**/
public double getTotArea(List listShapes, double dim){
    Iterator it = listShapes.iterator();
    double totalArea = 0.0;
    //loop through different shapes
    while(it.hasNext()) {
       Shape s =  (Shape) it.next();
       totalArea += s.area(dim); //polymorphic method call
    }
    return  totalArea ;
}
```

For example: given a base class/interface *Shape*, polymorphism allows the programmer to define different area(double dim1) methods for any number of derived classes such as *Circle*, *Square* etc. No matter what shape an object is, applying the area method to it will return the right results.

Later on *HalfCicle* can be added without breaking your called code i.e. method getTotalArea(...)

Depending on what the shape is, appropriate area(double dim) method gets called and calculated.

Circle → area is 78.5sqm
Square → area is 25sqm
HalfCircle → area is 39.25 sqm



Inheritance – is the inclusion of behaviour (i.e. methods) and state (i.e. variables) of a base class in a derived class so that they are accessible in that derived class. The key benefit of Inheritance is that it provides the formal mechanism for **code reuse**. Any shared piece of business logic can be moved from the derived class into the base class as part of refactoring process to improve maintainability of your code by avoiding code duplication. The existing class is called the *superclass* and the derived class is called the *subclass*. **Inheritance** can also be defined as the process whereby one object acquires characteristics from one or more other objects the same way children acquire characteristics from their parents.

There are two types of inheritances:

1. **Implementation inheritance** (aka class inheritance): You can extend an applications' functionality by reusing functionality in the parent class by inheriting all or some of the operations already implemented. In Java, you can only inherit from one superclass. Implementation inheritance promotes reusability but improper use of class inheritance can cause programming nightmares by breaking encapsulation and making future changes a problem. With implementation inheritance, the subclass becomes tightly coupled with the superclass. This will make the design fragile because if you want to change the superclass, you must know all the details of the subclasses to avoid breaking them. So when using implementation inheritance, **make sure that the subclasses depend only on the behaviour of the superclass, not on the actual implementation**. For example in the above diagram the subclasses should only be concerned about the behaviour known as area() but not how it is implemented.

2. **Interface inheritance** (aka type inheritance): This is also known as subtyping. Interfaces provide a mechanism for specifying a relationship between otherwise unrelated classes, typically by specifying a set of common methods each implementing class must contain. Interface inheritance promotes the design concept of **program to interfaces not to implementations**. This also reduces the coupling or implementation dependencies between systems. In Java, you can implement any number of interfaces. This is more flexible than implementation inheritance because it won't lock you into specific implementations which make subclasses difficult to maintain. So care should be taken not to break the implementing classes by modifying the interfaces.

Which one to use? Prefer interface inheritance to implementation inheritance because it promotes the design concept of **coding to an interface** and **reduces coupling**. Interface inheritance can achieve **code reuse** with the help of **object composition**. If you look at Gang of Four (GoF) design patterns, you can see that it favours interface inheritance to implementation inheritance. **CO**

Implementation inheritance

Let's assume that savings account and term deposit account have a similar behaviour in terms of depositing and withdrawing money, so we will get the super class to implement this behaviour and get the subclasses to reuse this behaviour. But saving account and term deposit account have specific behaviour in calculating the interest.

```
public abstract class Account {

  public void deposit(double amount) {
    //deposit logic
  }

  public void withdraw(double amount) {
    //withdraw logic
  }

  public abstract double calculateInterest(double  amount);

}
```

```
public class SavingsAccount extends Account {

   public double calculateInterest(double amount)  {
      //calculate interest for SavingsAccount
  }
}
```

```
public class TermDepositAccount extends Account {

    public double calculateInterest(double amount) {
      //calculate interest for TermDeposit
    }
 }
```

The calling code can be defined as follows for illustration purpose only:

```
public class Test {
  public static void main(String[] args) {
    Account acc1 = new SavingsAccount();
    acc1.deposit(5.0);
    acc1.withdraw(2.0);

    Account acc2 = new TermDepositAccount();
    acc2.deposit(10.0);
    acc2.withdraw(3.0);

    acc1.calculateInterest(500.00);
    acc2.calculateInterest(500.00);
  }
}
```

Interface inheritance

Let's look at an **interface inheritance** code sample, which makes use of **composition** for reusability. In the following example the methods deposit(…) and withdraw(…) share the same piece of code in *AccountHelper* class. The method calculateInterest(…) has its specific implementation in its own class.

```
public interface Account {
    public abstract void deposit(double amount);
    public abstract void withdraw(double amount);
    public abstract int getAccountType();
}

public interface SavingsAccount extends Account{
   public abstract double calculateInterest(double amount);
}

public interface TermDepositAccount extends Account{
   public abstract double calculateInterest(double amount);
}
```

The classes **SavingsAccountImpl, TermDepositAccountImpl** should implement the methods declared in its interfaces. The class **AccountHelper** implements the methods deposit(…) and withdraw(…)

```
public class SavingsAccountImpl implements SavingsAccount{
    private int accountType = 1;

   //helper class which promotes code reuse through composition
   AccountHelper helper = new AccountHelper();

   public void deposit(double amount) {
        helper.deposit(amount, getAccountType());
   }
   public void withdraw(double amount) {
        helper.withdraw(amount, getAccountType());
   }
   public double calculateInterest(double amount) {
        //calculate interest for SavingsAccount
    }
   public int getAccountType(){
       return accountType;
   }
}
```

```
public class TermDepositAccountImpl implements
                                        TermDepositAccount {
     private int accountType = 2;

    //helper class which promotes code reuse through composition
    AccountHelper helper = new AccountHelper();

    public void deposit(double amount) {
          helper.deposit(amount, getAccountType());
    }
    public void withdraw(double amount) {
          helper.withdraw(amount, getAccountType());
    }
    public double calculateInterest(double amount) {
           //calculate interest for TermDeposit
     }
    public int getAccountType() {
         return accountType;
    }
}
```

The calling code can be defined as follows for illustration purpose only:

```
public class Test {
    public static void main(String[] args) {
```

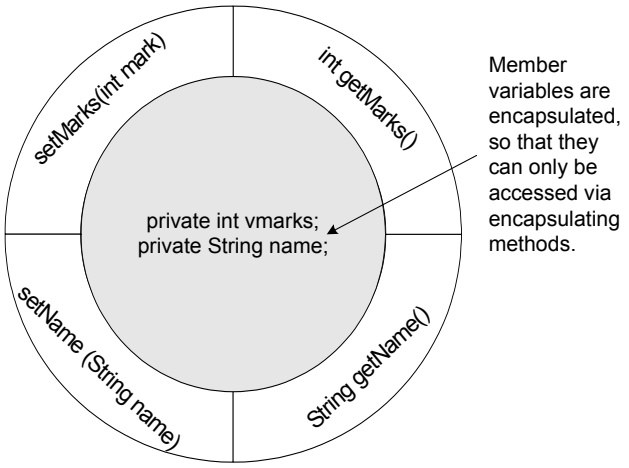
```
        Account acc1 = new SavingsAccountImpl();
        acc1.deposit(5.0);

        Account acc2 = new TermDepositAccountImpl();
        acc2.deposit(10.0);

        if (acc1.getAccountType() == 1) {
            ((SavingsAccount) acc1).calculateInterest(500.00);
        }

        if (acc2.getAccountType() == 2) {
            ((TermDepositAccount) acc2).calculateInterest(500.00);
        }
    }
}
```

Encapsulation – refers to keeping all the related members (variables and methods) together in an object. Specifying members as private can hide the variables and methods. Objects should hide their inner workings from the outside view. Good **encapsulation improves code modularity by preventing objects interacting with each other in an unexpected way**, which in turn makes future development and refactoring efforts easy.

Being able to encapsulate members of a class is important for **security** and **integrity**. We can protect variables from unacceptable values. The sample code below describes how encapsulation can be used to protect the *MyMarks* object from having negative values. Any modification to member variable "*vmarks*" can only be carried out through the setter method *setMarks(int mark)*. This prevents the object "*MyMarks*" from having any negative values by throwing an exception. **CO**

Sample code

```
Class MyMarks {
  private int vmarks =  0;
  private String name;

  public void setMarks(int mark)
                        throws MarkException {
      if(mark > 0)
         this.vmarks =  mark;
      else {
         throw new MarkException("No negative
                                        Values");
      }
  }

   public int getMarks(){
        return vmarks;
   }
   //getters and setters for attribute name goes here.
}
```



Q 09: What is design by contract? Explain the *assertion* construct? **DC**

A 09: Design by contract specifies the obligations of a calling-method and called-method to each other. Design by contract is a valuable technique, which should be used to build well-defined interfaces. The strength of this programming methodology is that it gets the programmer to **think clearly about what a function does**, what pre and post conditions it must adhere to and also it **provides documentation for the caller**. Java uses the ***assert*** statement to implement pre- and post-conditions. Java's exceptions handling also support design by contract especially **checked exceptions** (Refer **Q34** in Java section for checked exceptions). In design by contract in addition to specifying programming code to carrying out intended operations of a method the programmer also specifies:

1. Preconditions – This is the part of the contract the **calling-method must agree to**. Preconditions specify the conditions that must be true before a called method can execute. Preconditions involve the system state and the arguments passed into the method at the time of its invocation. **If a precondition fails then there is a bug in the calling-method or calling software component.**

On public methods	On non-public methods

Preconditions on *public* methods are enforced by explicit checks that throw particular, specified exceptions. You **should not use *assertion* to check the parameters of the public methods** but can use for the non-public methods. ***Assert*** is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled. Further, assert construct does not throw an exception of a specified type. It can throw only an *AssertionError*.
 	
`public void setRate(int rate) {`
`  if(rate <= 0 || rate > MAX_RATE){`
`  throw new IllegalArgumentException("Invalid rate → " + rate);`
`  }`
`  setCalculatedRate(rate);`
`}`
	You can use assertion to check the parameters of the non-public methods.
	
	`private void setCalculatedRate(int rate) {`
	`  assert (rate > 0 && rate < MAX_RATE) : rate;`
	`  //calculate the rate and set it.`
	`}`
	
	Assertions can be disabled, so programs must not assume that assert construct will be always executed:
	
	`//Wrong: if assertion is disabled, CarpenterJob never //Get removed`
	`assert jobsAd.remove(PilotJob);`
	
	`//Correct:`
	`boolean pilotJobRemoved = jobsAd.remove(PilotJob);`
	`assert pilotJobRemoved;`

2. ***Postconditions*** – This is the part of the contract the **called-method agrees to**. What must be true after a method completes successfully**.** Postconditions can be used with assertions in both public and non-public methods. The postconditions involve the old system state, the new system state, the method arguments and the method's return value. **If a postcondition fails then there is a bug in the called-method or called software component.**

```
public double calcRate(int rate) {
   if(rate <= 0 || rate > MAX_RATE){
      throw new IllegalArgumentException("Invalid rate !!! ");
   }

    //logic to calculate the rate and set it goes here

    assert this.evaluate(result) < 0 : this;   //this → message sent to AssertionError on failure
    return result;
}
```

3. ***Class invariants*** - what must be true about each instance of a class? A class invariant as an internal invariant that can specify the relationships among multiple attributes, and should be true before and after any method completes. **If an invariant fails then there could be a bug in either calling-method or called-method.** There is no particular mechanism for checking invariants but it is convenient to combine all the expressions required for checking invariants into a single internal method that can be called by assertions. For example if you have a class, which deals with negative integers then you define the **isNegative()** convenient internal method:

```
class NegativeInteger {
   Integer value = new Integer (-1); //invariant

   //constructor
   public NegativeInteger(Integer int) {
      //constructor logic goes here
      assert isNegative();
   }

   //rest of the public and non-public methods goes here. public methods should call assert isNegative(); prior to its return

   //convenient internal method for checking invariants. Returns true if the integer value is negative
   private boolean isNegative(){
      return  value.intValue() < 0 ;
    }
}
```

The isNegative() method should be true before and after any method completes, each public method and constructor should contain the following assert statement immediately prior to its return.

```
assert isNegative();
```

Explain the assertion construct? The assertion statements have two forms as shown below:

```
assert Expression1;
```

assert *Expression1* : *Expression2*;

Where:

- ***Expression1*** → is a boolean expression. If the *Expression1* evaluates to false, it throws an *AssertionError* without any detailed message.
- ***Expression2*** → if the *Expression1* evaluates to false throws an *AssertionError* with using the value of the *Expression2* as the errors' detailed message.

Note: If you are using assertions (available from JDK1.4 onwards), you should supply the JVM argument to enable it by package name or class name.

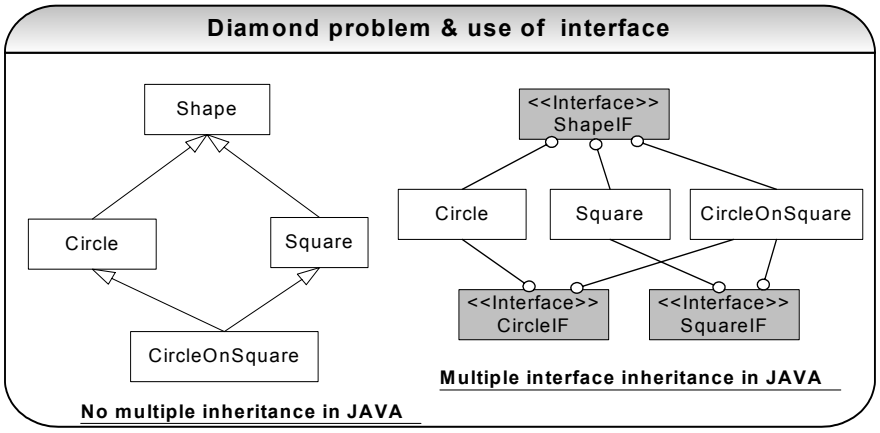
Java -ea[:packagename...|:classname] or **Java -enableassertions**[:packagename...|:classname]
Java –ea:Account

Q 10: What is the difference between an abstract class and an interface and when should you use them? **LF** **DP** **DC**

A 10: In design, you want the base class to present *only* an interface for its derived classes. This means, you don't want anyone to actually instantiate an object of the base class. You only **want to upcast to it** (implicit upcasting, which gives you polymorphic behaviour), so that its interface can be used. This is accomplished by making that class *abstract* using the **abstract** keyword. If anyone tries to make an object of an **abstract** class, the compiler prevents it.

The **interface** keyword takes this concept of an **abstract** class a step further by preventing any method or function implementation at all. You can only declare a method or function but not provide the implementation. The class, which is implementing the interface, should provide the actual implementation. The **interface** is a very useful and commonly used aspect in OO design, as it provides the **separation of interface and implementation** and enables you to:

- Capture similarities among unrelated classes without artificially forcing a class relationship.
- Declare methods that one or more classes are expected to implement.
- Reveal an object's programming interface without revealing its actual implementation.
- Model multiple interface inheritance in Java, which provides some of the benefits of full on multiple inheritances, a feature that some object-oriented languages support that allow a class to have more than one superclass.



Abstract class	Interface
Have executable methods and abstract methods.	Have no implementation code. All methods are abstract.
Can only subclass one abstract class.	A class can implement any number of interfaces.
Can have instance variables, constructors and any visibility: public, private, protected, none (aka package).	Cannot have instance variables, constructors and can have only public and none (aka package) visibility.

When to use an abstract class?: In case where you want to use **implementation inheritance** then it is usually provided by an abstract base class. Abstract classes are excellent candidates inside of application frameworks. Abstract classes let you define some default behaviour and force subclasses to provide any specific behaviour. Care should be taken not to overuse implementation inheritance as discussed in **Q8** in Java section.

When to use an interface?: For polymorphic interface inheritance, where the client wants to only deal with a type and does not care about the actual implementation use interfaces. If you need to change your design frequently, you should prefer using interface to abstract. **CO** Coding to an interface **reduces coupling** and interface inheritance can achieve **code reuse** with the help of **object composition**. Another justification for using interfaces is that they solve the '**diamond problem**' of traditional multiple inheritance as shown in the figure. Java does not support multiple inheritances. Java only supports **multiple interface inheritance**. Interface will solve all the ambiguities caused by this 'diamond problem'.

Design pattern: Strategy design pattern lets you swap new algorithms and processes into your program without altering the objects that use them. **Strategy design pattern**: Refer **Q11** in How would you go about… section.

Q 11: Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java? **LF**

A 11: The interfaces with no defined methods act like markers. They just tell the compiler that the objects of the classes implementing the interfaces with no defined methods need to be treated differently. ***Example*** Serializable (Refer **Q19** in Java section), Cloneable etc

---

Q 12: When is a method said to be overloaded and when is a method said to be overridden? **LF** **CO**

A 12:

Method Overloading	Method Overriding

Overloading deals with multiple methods in the same class with the same name but different method signatures.
 	
`class MyClass {`
`  public void getInvestAmount(int rate) {…}`
 	
`  public void getInvestAmount(int rate, long principal)`
`  { … }`
`}`
 	
Both the above methods have the same method names but different method signatures, which mean the methods are overloaded.
	Overriding deals with two methods, one in the parent class and the other one in the child class and has the same name and signatures.
	
	`class BaseClass{`
	`  public void getInvestAmount(int rate) {…}`
	`}`
	
	`class MyClass extends BaseClass {`
	`  public void getInvestAmount(int rate) { …}`
	`}`
	
	Both the above methods have the same method names and the signatures but the method in the subclass *MyClass* overrides the method in the superclass *BaseClass*.

Overloading lets you define the **same operation in different ways for different data**.	Overriding lets you define the **same operation in different ways for different object types**.

Q 13: What is the main difference between an ArrayList and a Vector? What is the main difference between Hashmap and Hashtable? **LF** **DC** **PI** **CI**

A 13:

Vector / Hashtable	ArrayList / Hashmap
Original classes before the introduction of Collections API. *Vector* & *Hashtable* are synchronized. Any method that touches their contents is thread-safe.	So if you don't need a thread safe collection, use the *ArrayList* or *Hashmap*. Why pay the price of synchronization unnecessarily at the expense of performance degradation.

So which is better? As a general rule, prefer *ArrayList/Hashmap* to *Vector/Hashtable*. If your application is a multithreaded application and **at least one of the threads either adds or deletes an entry into the collection** then use new Java c*ollection* API's external synchronization facility as shown below to **temporarily synchronize** your collections as needed: **CO**

```
Map myMap = Collections.synchronizedMap (myMap);
List  myList = Collections.synchronizedList (myList);
```

Java arrays are even faster than using an *ArrayList/Vector* and perhaps therefore may be preferable. *ArrayList/Vector* internally uses an array with some convenient methods like add(..), remove(…) etc.

Q 14: Explain the Java Collection framework? **LF** **DP**

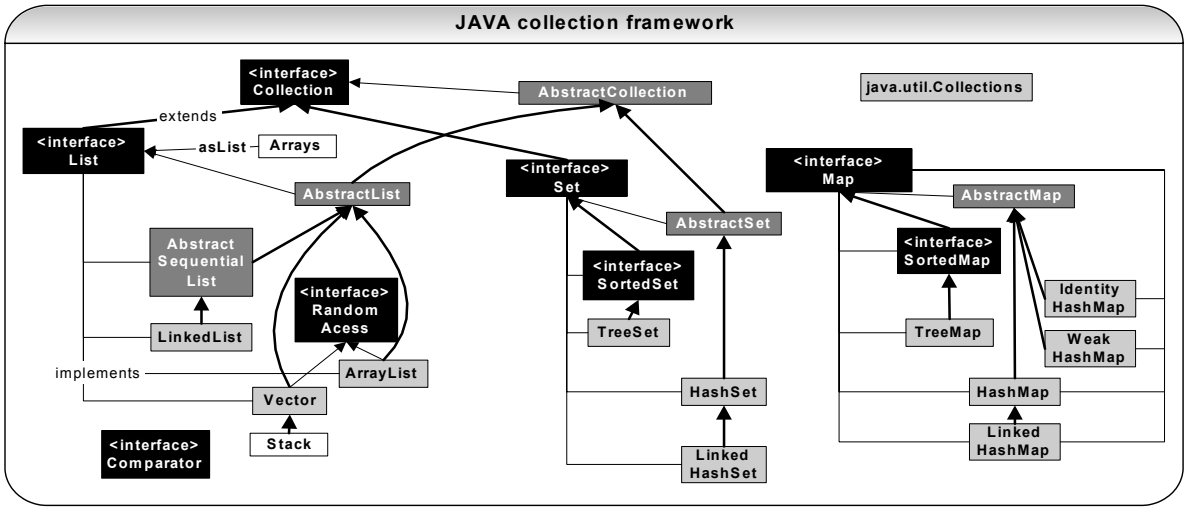
A 14: The key interfaces used by the collection framework are ***List***, ***Set*** and ***Map***. The *List* and *Set* extends the ***Collection*** interface. Should not confuse the *Collection* interface with the ***Collections*** class which is a utility class.

A ***Set*** is a collection with unique elements and prevents duplication within the collection. ***HashSet*** and ***TreeSet*** are implementations of a *Set* interface. A ***List*** is a collection with an ordered sequence of elements and may contain duplicates. ***ArrayList***, ***LinkedList*** and ***Vector*** are implementations of a *List* interface.

The Collection API also supports maps, but within a hierarchy distinct from the *Collection* interface. A ***Map*** is an object that maps keys to values, where the list of keys is itself a collection object. A map can contain duplicate values, but the keys in a map must be distinct. ***HashMap***, ***TreeMap*** and ***Hashtable*** are implementations of a *Map* interface.

How to implement collection ordering? ***SortedSet*** and ***SortedMap*** interfaces maintain sorted order. The classes, which implement the ***Comparable*** interface, impose natural order. For classes that don't implement comparable interface, or when one needs even more control over ordering based on multiple attributes, a ***Comparator*** interface should be used.

Design pattern: What is an Iterator? An Iterator is a use once object to access the objects stored in a collection. **Iterator design pattern** (aka Cursor) is used, which is a behavioural design pattern that provides a way to access elements of a collection sequentially without exposing its internal representation.



(**Diagram sourced from:** http://www.wilsonmar.com/1arrays.htm)

What are the benefits of the Java collection framework? Collection framework provides flexibility, performance, and robustness.

- **Polymorphic algorithms** – sorting, shuffling, reversing, binary search etc.
- **Set algebra** - such as finding subsets, intersections, and unions between objects.
- **Performance** - collections have much better performance compared to the older *Vector* and *Hashtable* classes with the elimination of synchronization overheads.
- **Thread-safety** - when synchronization is required, wrapper implementations are provided for temporarily synchronizing existing collection objects.
- **Immutability** - when immutability is required wrapper implementations are provided for making a collection immutable.
- **Extensibility** - interfaces and abstract classes provide an excellent starting point for adding functionality and features to create specialized object collections.

Q 15: What are some of the best practices relating to Java collection? **BP** **PI** **CI**

A 15:

- Use ArrayLists, HashMap etc as opposed to Vector, Hashtable etc, where possible to avoid any synchronization overhead. Even better is to use just arrays where possible. If multiple threads concurrently access a collection and **at least one of the threads either adds or deletes an entry into the collection**, then the collection must be externally synchronized. This is achieved by:

```
Map myMap = Collections.synchronizedMap (myMap);
```

```
List  myList = Collections.synchronizedList (myList);
```

- Set the initial capacity of a collection appropriately (e.g. ArrayList, HashMap etc). This is because collection classes like ArrayList, HashMap etc must grow periodically to accommodate new elements. But if you have a very large array, and you know the size in advance then you can speed things up by setting the initial size appropriately.

  For example: HashMaps/Hashtables need to be created with sufficiently large capacity to minimise **rehashing** (which happens every time the table grows). HashMap has two parameters initial capacity and load factor that affect its performance and space requirements. Higher load factor values (default load factor of 0.75 provides a good trade off between performance and space) will reduce the space cost but will increase the lookup cost of myMap.get(…) and myMap.put(…) methods. When the number of entries in the HashMap exceeds the **current capacity * loadfactor** then the capacity of the HasMap is roughly doubled by calling the rehash function. It is also very important not to set the initial capacity too high or load factor too low if iteration performance or reduction in space is important.

- **Program in terms of interface not implementation**: For example you might decide a LinkedList is the best choice for some application, but then later decide ArrayList might be a better choice for performance reason. **CO**

  Use:
  List list = new ArrayList(100); //program in terms of interface & set the initial capacity.
  Instead of:
  ArrayList list = new ArrayList();

- **Avoid storing unrelated or different types of objects into same collection**: This is analogous to storing items in pigeonholes without any labelling. To store items use **value objects** or **data objects** (as oppose to storing every attribute in an ArrayList or HashMap). Provide wrapper classes around your collection API classes like ArrayList, Hashmap etc as shown in better approach column. Also where applicable consider using **composite design pattern**, where an object may represent a single object or a collection of objects. Refer **Q52** in Java section for UML diagram of a composite design pattern. **CO**

Avoid where possible

The code below is hard to maintain and understand by others. Also gets more complicated as the requirements grow in the future because we are throwing different types of objects like Integer, String etc into a list just based on the indices and it is easy to make mistakes while casting the objects back during retrieval.

```
List myOrder = new ArrayList()

ResultSet rs = …

While (rs.hasNext()) {

    List lineItem = new ArrayList();

    lineItem.add (new Integer(rs.getInt("itemId")));
    lineItem.add (rs.getString("description"));
    ….
    myOrder.add( lineItem);
}

return myOrder;
```

Example 2:

```
List myOrder =  new ArrayList(10);

//create an order
OrderVO header = new OrderVO();
header.setOrderId(1001);
…
//add all the line items
LineItemVO line1 = new LineItemVO();
line1.setLineItemId(1);
LineItemVO line2 = new LineItemVO();
Line2.setLineItemId(2);
```

Better approach

When storing items into a collection define value objects as shown below: (**VO** is an acronym for **V**alue **O**bject).

```
public class LineItemVO {
  private int itemId;
  private String productName;

  public int getLineItemId(){return accountId ;}
  public int getAccountName(){return accountName;}

  public void setLineItemId(int accountId ){
    this.accountId = accountId
  }
   //implement other getter & setter methods
}
```

Now let's define our base wrapper class, which represents an order:

```
public abstract class Order {
      int orderId;
      List lineItems = null;

      public abstract int countLineItems();
      public abstract boolean add(LineItemVO itemToAdd);
      public abstract boolean remove(LineItemVO itemToAdd);
      public abstract Iterator  getIterator();
      public int  getOrderId(){return this.orderId; }
}
```

Now a specific implementation of our wrapper class:

```
public class OverseasOrder extends Order {
      public OverseasOrder(int inOrderId) {
            this.lineItems = new ArrayList(10);
            this.orderId = inOrderId;
      }
```

```
List lineItems = new ArrayList();
lineItems.add(line1);
lineItems.add(line2);

//to store objects
myOrder.add(order);// index 0 is an OrderVO object
myOrder.add(lineItems);//index 1 is a List of line items

//to retrieve objects
myOrder.get(0);
myOrder.get(1);
```

Above approaches are bad because disparate objects are stored in the **lineItem** collection in example-1 and example-2 relies on indices to store disparate objects. The indices based approach and storing disparate objects are hard to maintain and understand because indices are hard coded and get scattered across the code. If an index position changes for some reason, then you will have to change every occurrence, otherwise it breaks your application.

The above coding approaches are analogous to storing disparate items in a storage system without proper labelling and just relying on its grid position.

```
    public int countLineItems() { //logic to count }

    public boolean add(LineItemVO itemToAdd){
        …//additional logic or checks
        return  lineItems.add(itemToAdd);
    }

    public boolean remove(LineItemVO itemToAdd){
        return  lineItems.remove(itemToAdd);
    }

    public ListIterator getIterator(){ return lineItems.Iterator();}
}
```

Now to use:

```
Order myOrder = new OverseasOrder(1234) ;

LineItemVO item1 = new LineItemVO();
Item1.setItemId(1);
Item1.setProductName("BBQ");

LineItemVO item2 = new LineItemVO();
Item1.setItemId(2);
Item1.setProductName("Outdoor chair");

//to add line items to order
myOrder.add(item1);
myOrder.add(item2);
…
```

Q 16: When providing a user defined key class for storing objects in the Hashmaps or Hashtables, what methods do you have to provide or override (i.e. **method overriding**)? LF PI CO

A 16: You should override the **equals()** and **hashCode()** methods from the *Object* class. The default implementation of the equals() and hashcode(), which are inherited from the java.lang.Object uses an object instance's memory location (e.g. MyObject@6c60f2ea). This can cause problems when two instances of the car objects have the same colour but the inherited equals() will return false because it uses the memory location, which is different for the two instances. Also the **toString()** method can be overridden to provide a proper string representation of your object. ***Points to consider:***

- If a class overrides **equals()**, it must override **hashCode()**.
- If 2 objects are equal, then their hashCode values must be equal as well.
- If a field is not used in equals(), then it must not be used in hashCode().
- If it is accessed often, hashCode() is a candidate for caching to enhance performance.

Note: Java 1.5 introduces enumerated constants, which improves readability and maintainability of your code. Java programming language enums are more powerful than their counterparts in other languages. E.g. A class like *Weather* can be built on top of simple enum type *Season* and the class *Weather* can be made immutable, and only one instance of each *Weather can* be created, so that your *Weather* class **does not have to override equals()** and **hashCode()** methods.

```
public class Weather {
   public enum Season {WINTER, SPRING, SUMMER, FALL}
   private final Season season;
   private static final List<Weather> listWeather =  new ArrayList<Weather> ();

   private Weather (Season season) {   this.season = season;}
   public Season getSeason () { return season;}

   static {
      for (Season season : Season.values()) {
           listWeather.add(new Weather(season));
      }
  }

  public static ArrayList<Weather> getWeatherList () { return  listWeather; }
  public String toString(){ return season;}  // takes advantage of toString() method of Season.
}
```

Q 17: What is the main difference between a String and a StringBuffer class? LF PI CI CO

A 17:

String	StringBuffer / StringBuilder
String is **immutable**: you can't modify a string object but can replace it by creating a new instance. Creating a new instance is rather expensive.	StringBuffer is **mutable**: use StringBuffer or StringBuilder when you want to modify the contents. ***StringBuilder*** was added in Java 5 and it is identical in all respects to ***StringBuffer*** except that it is not synchronised, which makes it slightly faster at the cost of not being thread-safe.

//Inefficient version using immutable String
String output = "Some text"
Int count = 100;
for(int I =0; i<count; i++) {
output += i;
}
return output;
	//More efficient version using mutable StringBuffer
	StringBuffer output = new StringBuffer(**110**);
	Output.append("Some text");
	for(int I =0; i<count; i++) {
	output.append(i);
	}
	return output.toString();

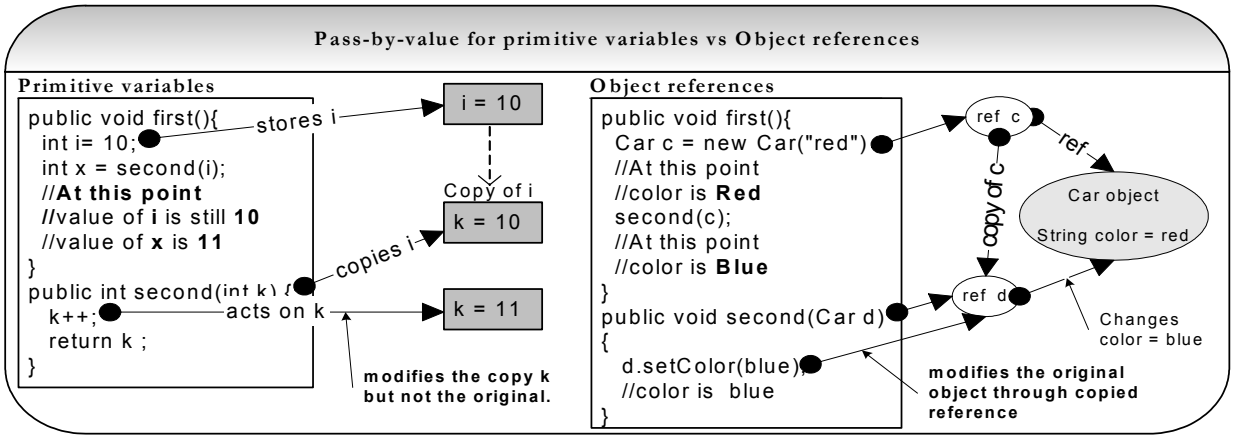
The above code would build 99 new String objects, of which 98 would be thrown away immediately. Creating new objects is not efficient.	The above code creates only two new objects, the *StringBuffer* and the final *String* that is returned. StringBuffer expands as needed, which is costly however, so it would be better to initilise the *StringBuffer* with the correct size from the start as shown.

Another important point is that creation of extra strings is not limited to 'overloaded mathematical operators' ("+") but there are several methods like **concat(), trim(), substring(),** and **replace()** in String classes that generate new string instances. So use StringBuffer or StringBuilder for computation intensive operations, which offer better performance.

Q 18: What is the main difference between pass-by-reference and pass-by-value? LF PI

A 18: Other languages use **pass-by-reference** or pass-by-pointer. But in Java no matter what type of argument you pass the corresponding parameter (primitive variable or object reference) will get a copy of that data, which is exactly how **pass-by-value** (i.e. copy-by-value) works.

In Java, if a calling method passes a reference of an object as an argument to the called method then the **passed-in reference gets copied first** and then passed to the called method. Both the original reference that was passed-in and the copied reference will be pointing to the same object. So no matter which reference you use, you will be always modifying the same original object, **which is how the pass-by-reference works as well**.

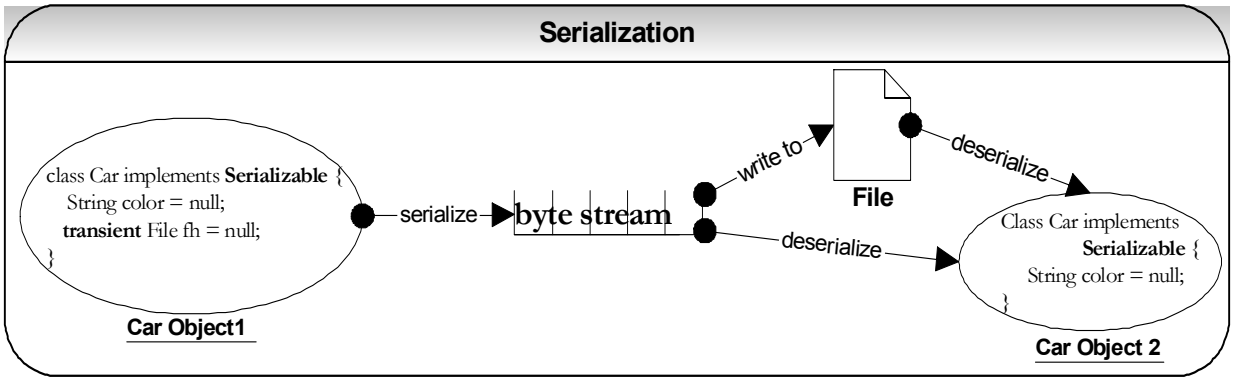


If your method call involves inter-process (e.g. between two JVMs) communication, then the reference of the calling method has a different address space to the called method sitting in a separate process (i.e. separate JVM). Hence inter-process communication involves calling method passing objects as arguments to called method by-value in a serialized form, which can adversely affect performance due to marshalling and unmarshalling cost.

Note: As discussed in **Q69** in Enterprise section, EJB 2.x introduced local interfaces, where enterprise beans that can be used locally within the same JVM using Java's form of **pass-by-reference**, hence improving performance.

Q 19: What is serialization? How would you exclude a field of a class from serialization or what is a transient variable? What is the common use? LF SI PI

A 19: Serialization is a process of reading or writing an object. It is a process of saving an object's state to a sequence of bytes, as well as a process of rebuilding those bytes back into a live object at some future time. An object is marked serializable by implementing the *java.io.Serializable* interface, which is only a *marker* interface -- it simply allows the serialization mechanism to verify that the class can be persisted, typically to a file.



Transient variables cannot be serialized. The fields marked **transient** in a serializable object will not be transmitted in the byte stream. An example would be a file handle or a database connection. Such objects are only meaningful locally. So they should be marked as transient in a serializable class.

Serialization can adversely affect performance since it:

- Depends on reflection.
- Has an incredibly verbose data format.
- Is very easy to send surplus data.

When to use serialization? Do not use serialization if you do not have to. A common use of serialization is to use it to send an object over the network or if the state of an object needs to be persisted to a flat file or a database. (Refer **Q57** on Enterprise section). Deep cloning or copy can be achieved through serialization. This may be fast to code but will have performance implications (Refer **Q22** in Java section).

The **objects stored in an HTTP session should be serializable** to support in-memory replication of sessions to achieve scalability (Refer **Q20** in Enterprise section). Objects are passed in RMI (Remote Method Invocation) across network using serialization (Refer **Q57** in Enterprise section).

Q 20: Explain the Java I/O streaming concept and the use of the decorator design pattern in Java I/O? LF DP PI SI

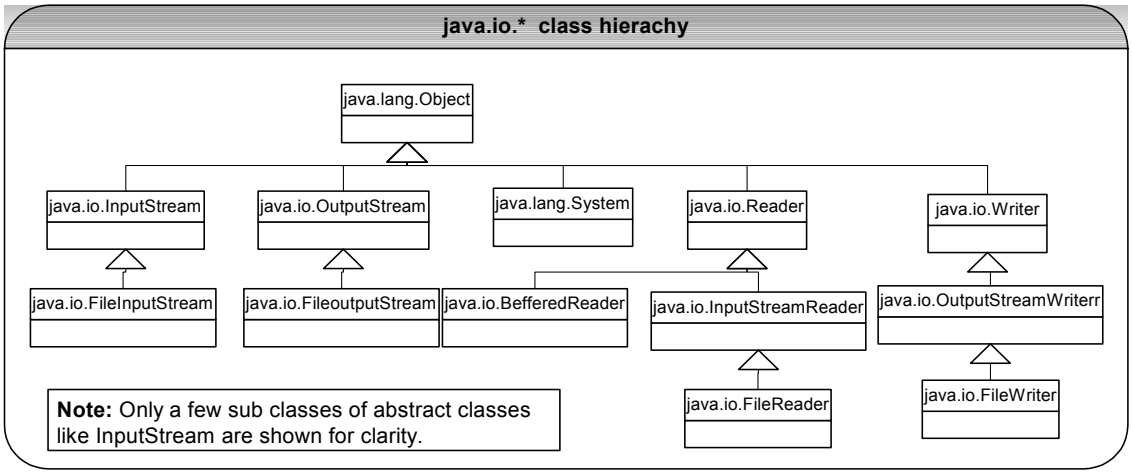
A 20: Java input and output is defined in terms of an abstract concept called a "**stream**", which is a sequence of data. There are 2 kinds of streams.

- Byte streams (8 bit bytes) → Abstract classes are: **InputStream** and **OutputStream**
- Character streams (16 bit UNICODE) → Abstract classes are: **Reader** and **Writer**

Design pattern: *java.io.** classes use the **decorator design pattern**. The decorator design pattern **attaches responsibilities to objects at runtime**. Decorators are more flexible than inheritance because the **inheritance attaches responsibility to classes at compile time**. The *java.io.** classes use the decorator pattern to construct different combinations of behaviour at runtime based on some basic classes.

Attaching responsibilities to classes at compile time using subclassing.	Attaching responsibilities to objects at runtime using a decorator design pattern.
Inheritance (aka subclassing) attaches responsibilities to classes at compile time. When you extend a class, each individual changes you make to child class will affect all instances of the child classes. Defining many classes using inheritance to have all possible combinations is problematic and inflexible.	By attaching responsibilities to **objects at runtime**, you can apply changes to each individual object you want to change. File **file** = new File("c:/temp"); FileInputStream **fis** = new FileInputStream(**file**); BufferedInputStream bis = new BufferedInputStream(**fis**); Decorators decorate an object by enhancing or restricting functionality of an object it decorates. The decorators add or restrict functionality to decorated

	objects either before or after forwarding the request. At runtime the BufferedInputStream (bis), which is a **decorator** (aka a **wrapper** around decorated object), forwards the method call to its **decorated** object FileInputStream (fis). The 'bis' will apply the additional functionality of buffering around the lower level file (i.e. fis) I/O.



The New I/O (NIO): more scalable and better performance

Java has long been not suited for developing programs that perform a lot of I/O operations. Furthermore, commonly needed tasks such as file locking, non-blocking and asynchronous I/O operations and ability to map file to memory were not available. Non-blocking I/O operations were achieved through work around such as multithreading or using JNI. The **New I/O** API (aka **NIO**) in J2SE 1.4 has changed this situation.

A server's ability to handle several client requests effectively depends on how it uses I/O streams. When a server has to handle hundreds of clients simultaneously, it must be able to use I/O services concurrently. One way to cater for this scenario in Java is to use threads but having almost one-to-one ratio of threads (100 clients will have 100 threads) is prone to enormous **thread overhead and can result in performance and scalability problems due to consumption of memory stacks and CPU context switching**. To overcome this problem, a new set of non-blocking I/O classes have been introduced to the Java platform in java.nio package. The non-blocking I/O mechanism is built around *Selectors* and *Channels*. ***Channels***, ***Buffers*** and ***Selectors*** are the core of the NIO.

A **Channel** class represents a bi-directional communication channel (similar to *InputStrean* and *OutputStream*) between datasources such as a socket, a file, or an application component, which is capable of performing one or more I/O operations such as reading or writing. Channels can be non-blocking, which means, no I/O operation will wait for data to be read or written to the network. The good thing about NIO channels is that they can be asynchronously interrupted and closed. So if a thread is blocked in an I/O operation on a channel, another thread can interrupt that blocked thread.

Buffers hold data. Channels can fill and drain *Buffers*. Buffers replace the need for you to do your own buffer management using byte arrays. There are different types of Buffers like ByteBuffer, CharBuffer, DoubleBuffer, etc.

A **Selector** class is responsible for multiplexing (combining multiple streams into a single stream) by allowing a single thread to service multiple channels. Each *Channel* registers events with a *Selector*. When events arrive from clients, the Selector demultiplexes (separating a single stream into multiple streams) them and dispatches the events to corresponding Channels. To achieve non-blocking I/O a *Channe*l class must work in conjunction with a *Selector* class.

Design pattern: NIO uses a **reactor design pattern**, which demultiplexes events (separating single stream into multiple streams) and dispatches them to registered object handlers. The reactor pattern is similar to an **observer pattern** (aka publisher and subscriber design pattern), but an observer pattern handles only a single source of events (i.e. a single publisher with multiple subscribers) where a reactor pattern handles multiple event sources (i.e. multiple publishers with multiple subscribers). The intent of an observer pattern is to define a one-to-many dependency so that when one object (i.e. the publisher) changes its state, all its dependents (i.e. all its subscribers) are notified and updated correspondingly.

Another sought after functionality of NIO is its ability to map a file to memory. There is a specialized form of a Buffer known as MappedByteBuffer, which represents a buffer of bytes mapped to a file. To map a file to

MappedByteBuffer, you must first get a channel for a file. Once you get a channel then you map it to a buffer and subsequently you can access it like any other ByteBuffer. Once you map an input file to a CharBuffer, you can do pattern matching on the file contents. This is similar to running "grep" on a UNIX file system.

Another feature of NIO is its ability to lock and unlock files. Locks can be exclusive or shared and can be held on a contiguous portion of a file. But file locks are subject to the control of the underlying operating system.

Q 21: How can you improve Java I/O performance? **PI** **BP**

A 21: Java applications that utilise Input/Output are excellent candidates for performance tuning. Profiling of Java applications that handle significant volumes of data will show significant time spent in I/O operations. This means substantial gains can be had from I/O performance tuning. Therefore, I/O efficiency should be a high priority for developers looking to optimally increase performance.

The basic rules for speeding up I/O performance are

- Minimise accessing the hard disk.
- Minimise accessing the underlying operating system.
- Minimise processing bytes and characters individually.

Let us look at some of the techniques to improve I/O performance. **CO**

- Use **buffering** to minimise disk access and underlying operating system. As shown below, with buffering large chunks of a file are read from a disk and then accessed a byte or character at a time.

Without buffering : inefficient code

```
try{
   File f = new File("myFile.txt");
   FileInputStream fis = new FileInputStream(f);
   int count = 0;
   int b = ;
   while((b = fis.read()) != -1){
     if(b== '\n') {
       count++;
     }
   }
    // fis should be closed in a finally block.
    fis.close() ;
 }
catch(IOException io){}
```

Note: fis.read() is a native method call to the underlying system.

With Buffering: yields better performance

```
try{
   File f = new File("myFile.txt");
   FileInputStream fis = new FileInputStream(f);
   BufferedInputStream bis = new BufferedInputStream(fis);
   int count = 0;
   int b = ;
   while((b = bis.read()) != -1){
     if(b== '\n') {
       count++;
     }
   }
    //bis should be closed in a finally block.
    bis.close() ;
 }
catch(IOException io){}
```

Note: bis.read() takes the next byte from the input buffer and only rarely access the underlying operating system.

Instead of reading a character or a byte at a time, the above code with buffering can be improved further by reading one line at a time as shown below:

```
FileReader fr = new FileReader(f);
BufferedReader br = new BufferedReader(fr);
While (br.readLine() != null) count++;
```

By default the **System.out** is line buffered, which means that the output buffer is flushed when a new line character is encountered. This is required for any interactivity between an input prompt and display of output. The line buffering can be disabled for faster I/O operation as follows:

```
FileOutputStream fos = new FileOutputStream(file);
BufferedOutputStream bos = new BufferedOutputStream(fos, 1024);
PrintStream ps  = new PrintStream(bos,false);
System.setOut(ps);

while (someConditionIsTrue)
        System.out.println("blah…blah…");
}
```

It is recommended to use logging frameworks like **Log4J** or **apache commons logging**, which uses buffering instead of using default behaviour of **System.out.println(…..)** for better performance. Frameworks like Log4J are configurable, flexible, extensible and easy to use.

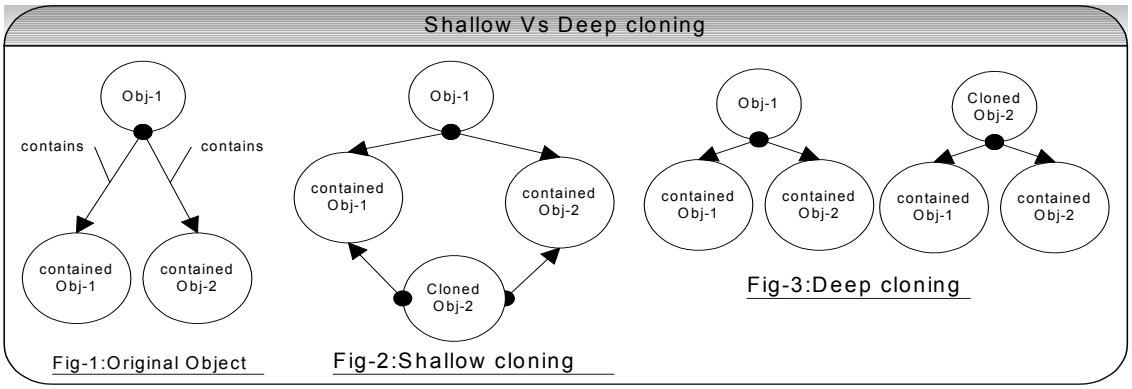
- Use the NIO package, if you are using JDK 1.4 or later, which uses performance-enhancing features like buffers to hold data, memory mapping of files, non-blocking I/O operations etc.
- I/O performance can be improved by minimising the calls to the underlying operating systems. The Java runtime itself cannot know the length of a file, querying the file system for isDirectory(), isFile(), exists() etc must query the underlying operating system.
- Where applicable caching can be used to improve performance by reading in all the lines of a file into a Java collection class like an ArrayList or a HashMap and subsequently access the data from an in-memory collection instead of the disk.

Q 22: What is the main difference between shallow cloning and deep cloning of objects? DC LF MI PI

A 22: The default behaviour of an object's clone() method automatically yields a shallow copy. So to achieve a deep copy the classes must be edited or adjusted.

Shallow copy: If a shallow copy is performed on obj-1 as shown in fig-2 then it is copied but its contained objects are not. The contained objects Obj-1 and Obj-2 are affected by changes to cloned Obj-2. Java supports shallow cloning of objects by default when a class implements the *java.lang.Cloneable* interface.

Deep copy: If a deep copy is performed on obj-1 as shown in fig-3 then not only obj-1 has been copied but the objects contained within it have been copied as well. Serialization can be used to achieve deep cloning. Deep cloning through serialization is faster to develop and easier to maintain but carries a performance overhead.



For example, invoking clone() method on a *HashMap* returns a shallow copy of *HashMap* instance, which means **the keys and values themselves are not cloned**. If you want a deep copy then a simple method is to serialize the *HashMap* to a *ByteArrayOutputSream* and then deserialize it. This creates a deep copy but does require that all keys and values in the *HashMap* are Serializable. Its primary advantage is that it will deep copy any arbitrary object graph.

List some of the methods supported by Java object class? clone(), toString(), equals(Object obj), hashCode() → refer **Q16** in Java section, wait(), notify() → refer **Q42** in Java section, finalize() etc.

Q 23: What is the difference between an instance variable and a static variable? Give an example where you might use a static variable? LF

A 23:

Static variable	Instance variable
Class variables are called static variables. There is only one occurrence of a class variable per JVM per class loader. When a class is loaded the class variables (aka static variables) are initialised.	Instance variables are non-static and there is one occurrence of an instance variable in each class instance (i.e. each object).

A static variable is used in the **singleton** pattern. (Refer **Q45** in Java section). A static variable is used with a **final** modifier to define **constants**.

Q 24: Give an example where you might use a static method? LF

A 24: Static methods prove useful for creating **utility classes, singleton classes** and **factory methods** (Refer **Q45**, **Q46** in Java section). Utility classes are not meant to be instantiated. Improper coding of utility classes can lead to procedural coding. **java.lang.Math, java.util.Collections** etc are examples of utility classes in Java.

Q 25: What are access modifiers? LF

A 25:

Modifier	Used with	Description
public	Outer classes, interfaces, constructors, Inner classes, methods and field variables	A class or interface may be accessed from outside the package. Constructors, inner classes, methods and field variables may be accessed wherever their class is accessed.
protected	Constructors, inner classes, methods, and field variables.	Accessed by other classes in the same package or any subclasses of the class in which they are referred (i.e. **same package** or **different package**).
private	Constructors, inner classes, methods and field variables,	Accessed only within the class in which they are declared
No modifier: (Package by default).	Outer classes, inner classes, interfaces, constructors, methods, and field variables	Accessed only from within the package in which they are declared.

Q 26: Where and how can you use a private constructor? LF

A 26: Private constructor is used if you do not want other classes to instantiate the object. The instantiation is done by a public static method within the same class.

- Used in the singleton pattern. (Refer **Q45** in Java section).
- Used in the factory method pattern (Refer **Q46** in Java section).
- Used in utility classes e.g. StringUtils etc.

Q 27: What is a final modifier? Explain other Java modifiers? LF

A 27: A final class can't be extended i.e. A final class may not be subclassed. A final method can't be overridden when its class is inherited. You can't change value of a final variable (i.e. it is a constant).

Modifier	Class	Method	Property
static	A static inner class is just an inner class associated with the class, rather than with an instance.	cannot be instantiated, are called by classname.method, can only access static variables	Only one instance of the variable exists.
abstract	Cannot be instantiated, must be a superclass, used whenever one or more methods are abstract.	Method is defined but contains no implementation code (implementation code is included in the subclass). If a method is abstract then the entire class must be abstract.	N/A
synchronized	N/A	Acquires a **lock on the class for static methods**. Acquires a **lock on the instance for non-static** methods.	N/A
transient	N/A	N/A	Field should not be serialized.
final	Class cannot be inherited	Method cannot be overridden	Makes the variable a constant.
native	N/A	Platform dependent. No body, only signature.	N/A

Note: Be prepared for tricky questions on modifiers like, what is a "**volatile**"? Or what is a "**const**"? Etc. The reason it is tricky is that Java does have these keywords "const" and "volatile" as reserved, which means you can't name your variables with these names **but modifier "const" is not yet added in the language** and the **modifier "volatile" is very rarely used**.

The "volatile" modifier is used on member variables that may be modified simultaneously by other threads. Since other threads cannot see local variables, there is no need to mark local variables as volatile. E.g. **volatile** int number; **volatile** private List listItems = null; etc. The modifier volatile only synchronizes the variable marked as volatile whereas "synchronized" modifier synchronizes all variables.

Java uses the final modifier to declare constants. A final variable or constant declared as "final" has a value that is immutable and cannot be modified to refer to any other objects other than one it was initialized to refer to. So the "final" modifier applies only to the value of the variable itself, and not to the object referenced by the variable. This is where the "const" modifier can come in very **useful if added to the Java language**. A reference variable or a constant marked as "const" refers to an immutable object that cannot be modified. The reference variable itself can be modified, if it is not marked as "final". The "const" modifier will be applicable only to non-primitive types. The primitive types should continue to use the modifier "final".

Q 28: What is the difference between final, finally and finalize() in Java? LF

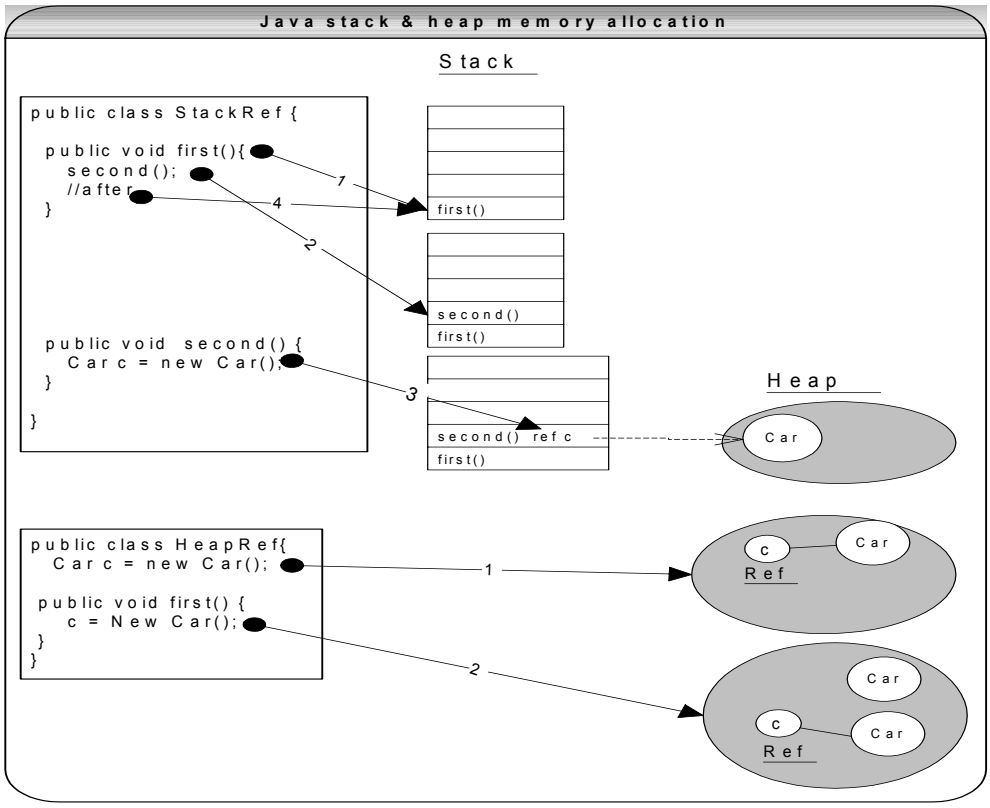
A 28:

- **final** - constant declaration. Refer **Q27** in Java section.
- **finally** - handles exception. The finally block is optional and provides a mechanism to clean up regardless of what happens within the try block (except System.exit(0) call). Use the finally block to close files or to release other system resources like database connections, statements etc. (Refer **Q45** in Enterprise section)
- **finalize()** - method helps in garbage collection. A **method** that is invoked before an object is discarded by the garbage collector, allowing it to clean up its state. Should not be used to release non-memory resources like file handles, sockets, database connections etc because Java has only a finite number of these resources and you do not know when the garbage collection is going to kick in to release these non-memory resources through the finalize() method.

Q 29: How does Java allocate stack and heap memory? Explain re-entrant, recursive and idempotent methods/functions? MI CI

A 29: Each time an object is created in Java it goes into the area of memory known as **heap**. The primitive variables like int and double are allocated in the **stack**, if they are local method variables and in the **heap** if they are member variables (i.e. fields of a class). In Java methods local variables are pushed into stack when a method is invoked and stack pointer is decremented when a method call is completed. In a multi-threaded application each thread will have its own stack but will share the same heap. This is why care should be taken in your code to avoid any concurrent access issues in the heap space. The stack is threadsafe (each thread will have its own stack) but the heap is not threadsafe unless guarded with synchronisation through your code.

A method in stack is **re-entrant** allowing multiple concurrent invocations that do not interfere with each other. A function is **recursive** if it calls itself. Given enough stack space, recursive method calls are perfectly valid in Java though it is tough to debug. Recursive functions are useful in removing iterations from many sorts of algorithms. All recursive functions are re-entrant but not all re-entrant functions are recursive. **Idempotent** methods are methods, which are written in such a way that repeated calls to the same method with the same arguments yield same results. For example clustered EJBs, which are written with idempotent methods, can automatically recover from a server failure as long as it can reach another server.



Q 30: Explain Outer and Inner classes (or Nested classes) in Java? When will you use an Inner Class? LF

A 30: In Java not all classes have to be defined separate from each other. You can put the definition of one class inside the definition of another class. The inside class is called an inner class and the enclosing class is called an outer class. So when you define an inner class, it is a member of the outer class in much the same way as other members like attributes, methods and constructors.

Where should you use inner classes? Code **without** inner classes **is more maintainable** and **readable**. When you access private data members of the outer class, the JDK compiler creates package-access member functions in the outer class for the inner class to access the private members. This leaves a **security hole**. In general **we should avoid using inner classes**. Use inner class only when an inner class is only relevant in the context of the outer class and/or inner class can be made private so that only outer class can access it. Inner classes are used primarily to implement helper classes like Iterators, Comparators etc which are used in the context of an outer class. **CO**

Member inner class

```
public class MyStack {
   private Object[] items = null;
   …
   public Iterator iterator() {
      return new StackIterator();
   }
   //inner class
   class StackIterator implements Iterator{
      …
      public boolean hasNext(){…}
   }
}
```

Anonymous inner class

```
public class MyStack {
   private Object[] items = null;
   …
   public Iterator iterator() {
      return new Iterator {
         …
         public boolean hasNext() {…}
      }
   }
}
```

Explain outer and inner classes?

Class Type		Description	Example + Class name
Outer class	Package member class or interface	Top level class. Only type JVM can recognize.	//package scope **class Outside{}** Outside.class

Inner class	static nested class or interface	Defined within the context of the top-level class. Must be static & can access static members of its containing class. No relationship between the instances of outside and Inside classes.	//package scope
			class Outside {
			static class Inside{ }
			}
			
			Outside.class ,Outside$Inside.class

Inner class	Member class	Defined within the context of outer class, but non-static. Until an object of Outside class has been created you can't create Inside.	class Outside{
			class Inside(){}
			}
			
			Outside.class , Outside$Inside.class

Inner class	Local class	Defined within a block of code. Can use final local variables and final method parameters. Only visible within the block of code that defines it.	class Outside {
			void first() {
			final int i = 5;
			class Inside{}
			}
			}
			
			Outside.class , Outside1Inside.class

Inner class	Anonymous class	Just like local class, but no name is used. Useful when only one instance is used in a method. Most commonly used in AWT event model.	class Outside{
			void first() {
			button.addActionListener **(new ActionListener()**
			{
			public void actionPerformed(ActionEvent e) {
			System.out.println("The button was pressed!");
			}
			});
			}
			}
			
			Outside.class , Outside$1.class

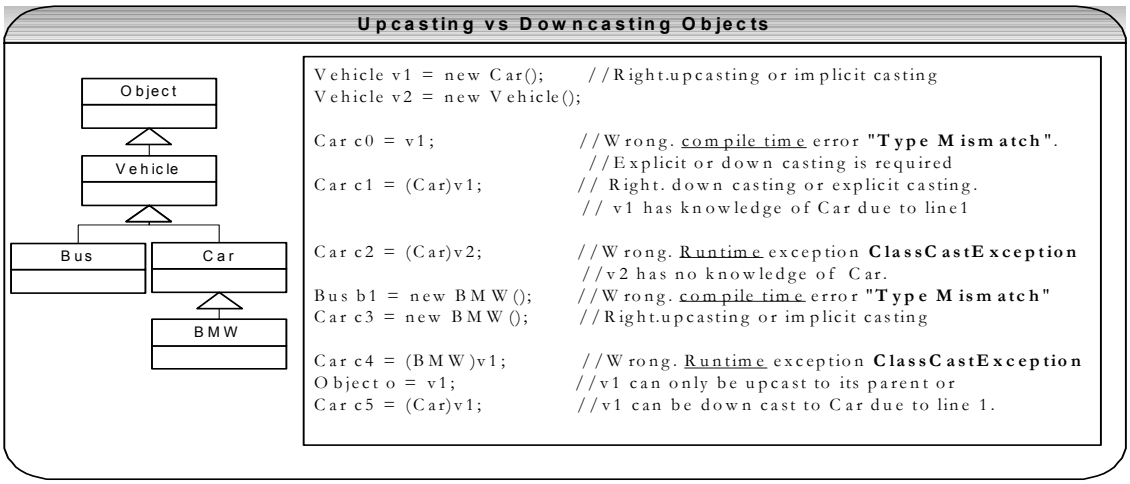
Q 31: What is type casting? Explain up casting vs. down casting? When do you get ClassCastException? **LF** **DP**

A 31: Type casting means treating a variable of one type as though it is another type.

When up casting **primitives** as shown below from left to right, automatic conversion occurs. But if you go from right to left, down casting or explicit casting is required. Casting in Java is safer than in C or other languages that allow arbitrary casting. Java only lets casts occur when they make sense, such as a cast between a float and an int. However you can't cast between an int and a *String* (is an object in Java).

```
byte → short → int → long → float → double

int i = 5;
long j = i;          //Right. Up casting or implicit casting
byte b1 = i;         //Wrong. Compile time error "Type Mismatch".
byte b2 = (byte) i ; //Right. Down casting or explicit casting is required.
```

When it comes to object references you can always cast from a subclass to a superclass because a subclass object is also a superclass object. You can cast an object implicitly to a super class type (i.e. **upcasting**). If this were not the case **polymorphism wouldn't be possible**.



You can cast down the hierarchy as well but you must explicitly write the cast and the **object must be a legitimate instance of the class you are casting to**. The **ClassCastException** is thrown to indicate that code has attempted to cast an object to a subclass of which it is not an instance. We can deal with the problem of incorrect casting in two ways:

- Use the exception handling mechanism to catch **ClassCastException**.

```
try{
        Object o = new Integer(1);
        System.out.println((String) o);
}
catch(ClassCastException cce) {
      logger.log("Invalid casting, String is expected…Not an Integer");
      System.out.println(((Integer) o).toString());
}
```

- Use the **instanceof** statement to guard against incorrect casting.

```
If(v2 instanceof Car) {
  Car c2 = (Car) v2;
}
```

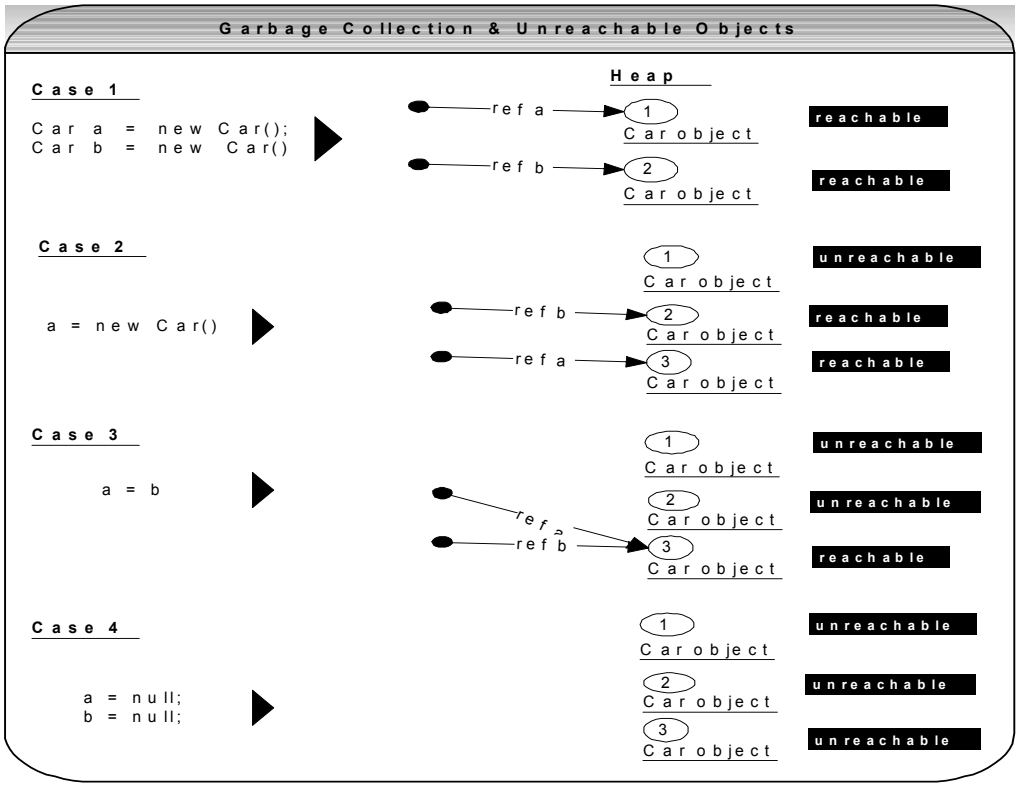
Design pattern: The "**instanceof**" and "**typecast"** constructs are shown for the illustration purpose only. Using these constructs can be unmaintainable due to large if and elseif statements and can affect performance if used in frequently accessed methods or loops. Look at using **visitor design pattern** to avoid these constructs. (Refer **Q11** in How would you go about section…).

Points-to-ponder: We can also get a **ClassCastException** when two different class loaders load the same class because they are treated as two different classes.

Q 32: What do you know about the Java garbage collector? When does the garbage collection occur? Explain different types of references in Java? LF MI

A 32: Each time an object is created in Java, it goes into the area of memory known as heap. The Java heap is called the garbage collectable heap. The garbage collection **cannot be forced**. The garbage collector runs in low memory situations. When it runs, it releases the memory allocated by an unreachable object. The garbage collector runs on a low priority daemon (background) thread. You can **nicely ask** the garbage collector to collect garbage by calling *System.gc()* but **you can't force it**.

What is an unreachable object? An object's life has no meaning unless something has reference to it. If you can't reach it then you can't ask it to do anything. Then the object becomes unreachable and the garbage collector will figure it out. Java automatically collects all the unreachable objects periodically and releases the memory consumed by those unreachable objects to be used by the future reachable objects.



We can use the following options with the **Java** command to enable tracing for garbage collection events.

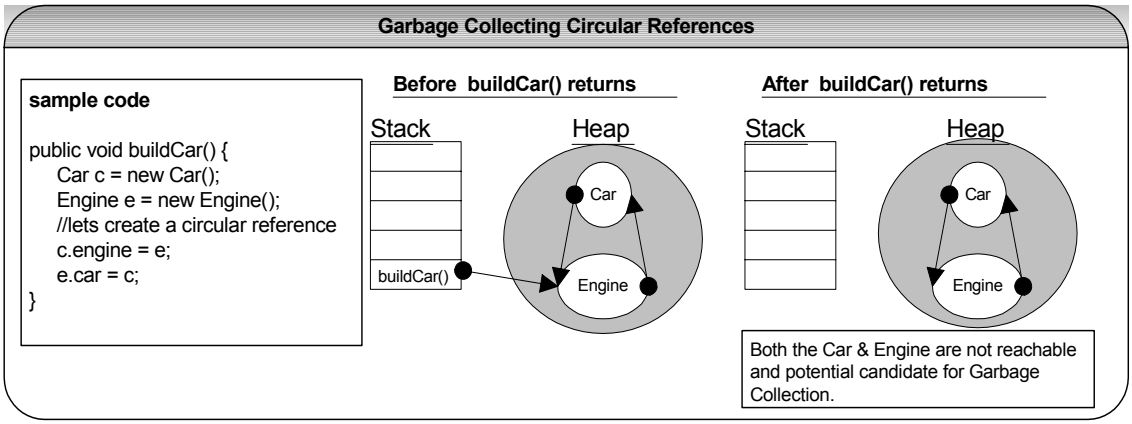
-verbose:gc reports on each garbage collection event.

Explain types of references in Java? java.lang.ref package can be used to declare soft, weak and phantom references.

- Garbage Collector won't remove a **strong reference**.
- A ***soft reference will*** only get removed if memory is low. So it is useful for implementing caches while avoiding memory leaks.
- A ***weak reference*** will get removed on the next garbage collection cycle. Can be used for implementing canonical maps. The **java.util.WeakHashMap** implements a *HashMap* with keys held by weak references.
- A ***phantom reference*** will be finalized but the memory will not be reclaimed. Can be useful when you want to be notified that an object is about to be collected.

Q 33: If you have a circular reference of objects, but you no longer reference it from an execution thread, will this object be a potential candidate for garbage collection? LF MI

A 33: Yes. Refer diagram below.

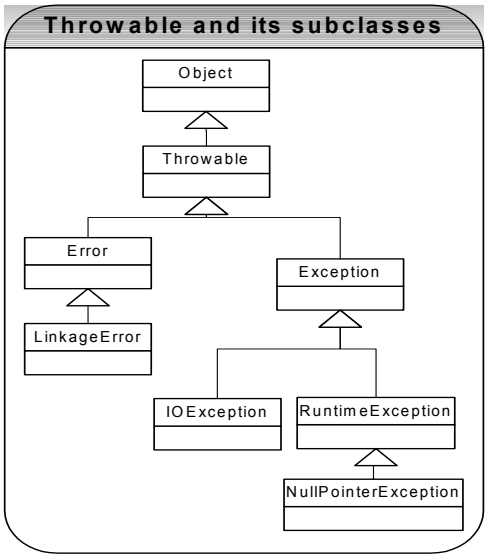


Q 34: Discuss the Java error handling mechanism? What is the difference between Runtime (**unchecked**) exceptions and **checked** exceptions? What is the implication of catching all the exceptions with the type "*Exception*"? EH BP

A 34:

Errors: When a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an Error. Typical Java programs should not catch Errors. In addition, it's unlikely that typical Java programs will ever throw Errors either.

Exceptions: Most programs throw and catch objects that derive from the Exception class. Exceptions indicate that a problem occurred but that the problem is not a serious JVM problem. An Exception class has many subclasses. These descendants indicate various types of exceptions that can occur. For example, NegativeArraySizeException indicates that a program attempted to create an array with a negative size. One exception subclass has special meaning in the Java language: RuntimeException. All the exceptions except RuntimeException are compiler checked exceptions. If a method is capable of throwing a checked exception it must declare it in its method header or handle it in a try/catch block. Failure to do so raises a compiler error. So checked exceptions can, at compile time, greatly reduce the occurrence of unhandled exceptions surfacing at runtime in a given application at the expense of requiring large throws declarations and encouraging use of poorly-constructed try/catch blocks. Checked exceptions are present in other languages like C++, C#, and Python.



Runtime Exceptions (unchecked exception)

A RuntimeException class represents exceptions that occur within the Java virtual machine (during runtime). An example of a runtime exception is NullPointerException. The cost of checking for the runtime exception often outweighs the benefit of catching it. Attempting to catch or specify all of them all the time would make your code unreadable and unmaintainable. The compiler allows runtime exceptions to go uncaught and unspecified. If you

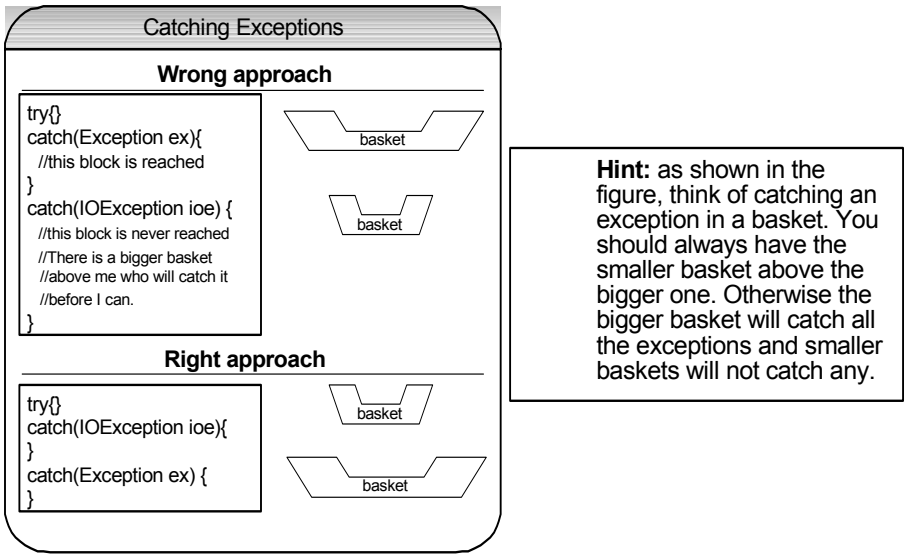
like, you can catch these exceptions just like other exceptions. However, you do not have to declare it in your "throws" clause or catch it in your catch clause. In addition, you can create your own *RuntimeException* subclasses and this approach is probably preferred at times because checked exceptions can complicate method signatures and can be difficult to follow.

Exception handling best practices: BP

Why is it not advisable to catch type "*Exception*"? CO

Exception handling in Java is **polymorphic** in nature. For example if you catch type *Exception in* your code then it can catch or throw its descendent types like *IOException* as well. So if you catch the type *Exception* before the type *IOException* then the type *Exception* block will catch the entire exceptions and type *IOException* block is never reached. In order to catch the type *IOException* and handle it differently to type *Exception*, *IOException* should be caught first (remember that you can't have a bigger basket above a smaller basket).

The diagram below is an example for illustration only. In practice it is not recommended to catch type "***Exception"***. We should only catch specific subtypes of the *Exception* class. Having a bigger basket (i.e. *Exception*) will hide or cause problems. Since the *RunTimeException* is a subtype of *Exception,* catching the type *Exception* will catch all the run time exceptions (like NullpointerException, ArrayIndexOut-OfBounds-Exception) as well.



Why should you throw an exception early? CO

The exception stack trace helps you pinpoint where an exception occurred by showing us the exact sequence of method calls that lead to the exception. By throwing your exception early, the exception becomes more accurate and more specific. Avoid suppressing or ignoring exceptions. Also avoid using exceptions just to get a flow control.

Instead of:

```
…
InputStream in = new FileInputStream(fileName); // assume this line throws an exception because filename == null.
…
```

Use the following code because you get a more accurate stack trace:

```
…
if(filename == null) {
    throw new IllegalArgumentException("file name is null");
}

InputStream in = new FileInputStream(fileName);
…
```

Why should you catch a checked exception late in a catch {} block?

You should not try to catch the exception before your program can handle it in an appropriate manner. The natural tendency when a compiler complains about a checked exception is to catch it so that the compiler stops reporting

errors. The best practice is to catch the exception at the appropriate layer (e.g. an exception thrown at an integration layer can be caught at a presentation layer in a catch {} block), where your program can either meaningfully recover from the exception and continue to execute or log the exception only once in detail, so that user can identify the cause of the exception.

Note: Due to heavy use of checked exceptions and minimal use of unchecked exceptions, there has been a hot debate in the Java community regarding true value of checked exceptions. Use checked exceptions when the client code can take some useful recovery action based on information in exception. Use unchecked exception when client code cannot do anything. For example, convert your SQLException into another checked exception if the client code can recover from it and convert your SQLException into an unchecked (i.e. RuntimeException) exception, if the client code cannot do anything about it.

A note on key words for error handling:

throw / throws – used to pass an exception to the method that called it.
try – block of code will be tried but may cause an exception.
catch – declares the block of code, which handles the exception.
finally – block of code, which is always executed (except System.exit(0) call) no matter what program flow, occurs when dealing with an exception.
assert – Evaluates a conditional expression to verify the programmer's assumption.

Q 35: What is a user defined exception? EH
A 35: User defined exceptions may be implemented by defining a new exception class by extending the *Exception* class.

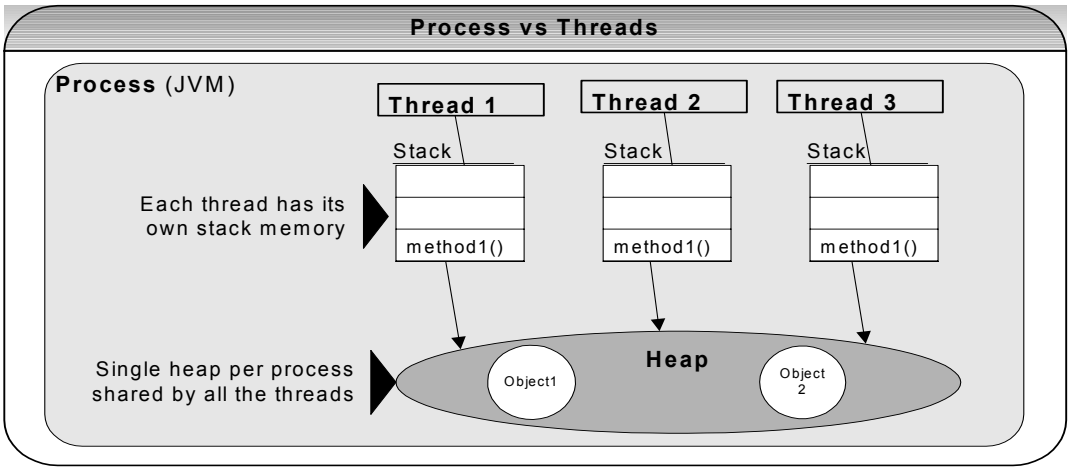
```
public class MyException extends Exception {

  /* class definition of constructors goes here */
  public MyException() {
    super();
  }

  public MyException (String errorMessage) {
    super (errorMessage);
  }
}
```

Throw and/or throws statement is used to signal the occurrence of an exception. Throw an exception:

```
throw new MyException("I threw my own exception.")
```

To declare an exception: `public myMethod() throws MyException {…}`

Q 36: What is the difference between processes and threads? LF MI CI
A 36: A process is an execution of a program but a thread is a single execution sequence within the process. A process can contain multiple threads. A thread is sometimes called a lightweight process.



A JVM runs in a single process and threads in a JVM share the heap belonging to that process. That is why several threads may access the same object. Threads **share the heap and have their own stack space**. This is

how one thread's invocation of a method and its local variables are kept thread safe from other threads. But the heap is not thread-safe and must be synchronized for thread safety.

Q 37: Explain different ways of creating a thread? LF

A 37: Threads can be used by either :

- Extending the ***Thread*** class
- Implementing the ***Runnable*** interface.

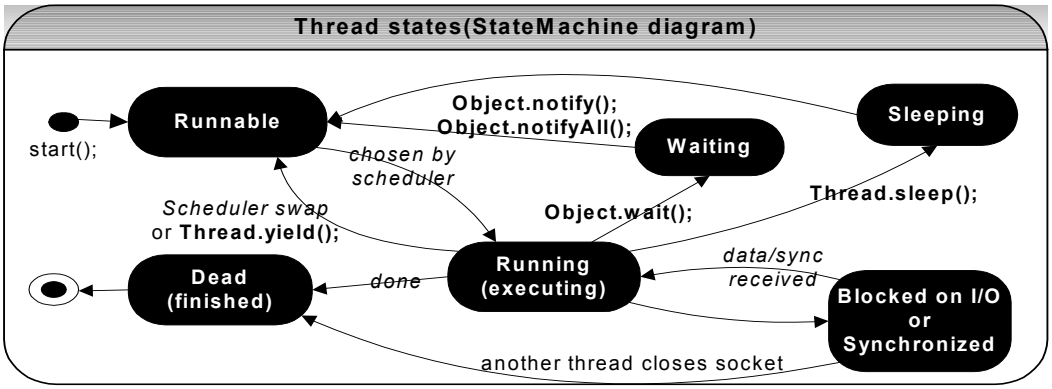
```
class Counter extends Thread {

   //method where the thread execution will start
   public void run(){
      //logic to execute in a thread
   }

   //let's see how to start the threads
   public static void main(String[] args){
        Thread t1 = new Counter();
        Thread t2 = new Counter();
         t1.start();  //start the first thread. This calls the run() method
         t2.start(); //this starts the 2nd thread. This calls the run() method
   }
}
```

```
class Counter extends Base implements Runnable {

   //method where the thread execution will start
   public void run(){
       //logic to execute in a thread
   }

   //let us see how to start the threads
   public static void main(String[] args){
        Thread t1 = new Thread(new Counter());
        Thread t2 = new Thread(new Counter());
         t1.start();  //start the first thread. This calls the run() method
         t2.start(); //this starts the 2nd thread. This calls the run() method
   }
}
```

The runnable interface is preferred, as it does not require your object to inherit a thread because when you need multiple inheritance, only interfaces can help you. In the above example we had to extend the *Base* class so implementing runnable interface is an obvious choice. Also note how the threads are started in each of the different cases as shown in the code sample.

Q 38: Briefly explain high-level thread states? LF

A 38: The state chart diagram below describes the thread states. (Refer **Q107** in Enterprise section for state chart diagram).



(**Diagram sourced from:** http://www.wilsonmar.com/1threads.htm)

- **Runnable** — waiting for its turn to be picked for execution by the thread schedular based on thread priorities.
- **Running**: The processor is actively executing the thread code. It runs until it becomes blocked, or voluntarily gives up its turn with this static method *Thread.yield()*. Because of context switching overhead, *yield()* should not be used very frequently.
- **Waiting**: A thread is in a **blocked state** while it waits for some external processing such as file I/O to finish.
- **Sleeping**: Java threads are forcibly put to sleep (suspended) with this overloaded method: Thread.sleep(milliseconds), Thread.sleep(milliseconds, nanoseconds);
- **Blocked on I/O**: Will move to runnable after I/O condition like reading bytes of data etc changes.
- **Blocked on synchronization**: Will move to Runnable when a **lock is acquired.**
- **Dead**: The thread is finished working.

Q 39: What is the difference between yield and sleeping? LF

A 39: When a task invokes yield(), it changes from running state to runnable state. When a task invokes sleep(), it changes from running state to waiting/sleeping state.

Q 40: How does thread synchronization occurs inside a monitor? What levels of synchronization can you apply? What is the difference between synchronized method and synchronized block? LF CI PI

A 40: In Java programming, each object has a lock. A thread can acquire the lock for an object by using the ***synchronized*** keyword. The synchronized keyword can be applied in **method level** (coarse grained lock – can affect performance adversely) or **block level of code** (fine grained lock). Often using a lock on a method level is too coarse. Why lock up a piece of code that does not access any shared resources by locking up an entire method. Since each object has a lock, dummy objects can be created to implement block level synchronization. The block level is more efficient because it does not lock the whole method.

```
class MethodLevel {
    //shared among threads
    SharedResource  x, y ;

    pubic void synchronized
method1() {
       //multiple threads can't access
    }

    pubic void synchronized
method2() {
       //multiple threads can't access
    }

    public void method3() {
      //not synchronized
      //multiple threads can access
    }
}
```

```
class BlockLevel {
    //shared among threads
    SharedResource  x, y ;
    //dummy objects for locking
    Object  xLock = new Object(), yLock = new Object();

    pubic void  method1() {
       synchronized(xLock){
          //access x here. thread safe
     }
       //do something here but don't  use
SharedResource  x, y ;

        synchronized(xLock) {
          synchronized(yLock) {
              //access x,y here. thread safe
           }
         }
       //do something here but don't use
SharedResource  x, y ;
    }
}
```

The JVM uses locks in conjunction with monitors. A monitor is basically a guardian who watches over a sequence of synchronized code and making sure only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code it must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock.

Once it has obtained the lock, the thread enters the block of protected code. When the thread leaves the block, no matter how it leaves the block, it releases the lock on the associated object.

Why synchronization is important? Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value. This often causes dirty data and leads to significant errors. **The disadvantage of synchronization** is that it can cause deadlocks when two threads are waiting on each other to do something. Also synchronized code has the overhead of acquiring lock, which can adversely the performance.

Q 41: What is a daemon thread? LF

A 41: Daemon threads are sometimes called "service" threads. These are threads that normally run at a low priority and provide a basic service to a program or programs when activity on a machine is reduced. An example of a daemon thread that is continuously running is the garbage collector thread. This thread is provided by the JVM.

Q 42: How can threads communicate with each other? How would you implement a producer (one thread) and a consumer (another thread) passing data (via stack)? LF

A 42: The **wait(), notify()**, and **notifyAll()** methods are used to provide an efficient way for threads to communicate with each other. This communication solves the '**consumer-producer problem**'. This problem occurs when the producer thread is completing work that the other thread (consumer thread) will use.

Example: If you imagine an application in which one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file. In this example the concurrent threads share the same resource file. Because these threads share the common resource file they should be synchronized. Also these two threads should communicate with each other because the consumer thread, which reads the file, should wait until the producer thread, which writes data to the file and notifies the consumer thread that it has completed its writing operation.

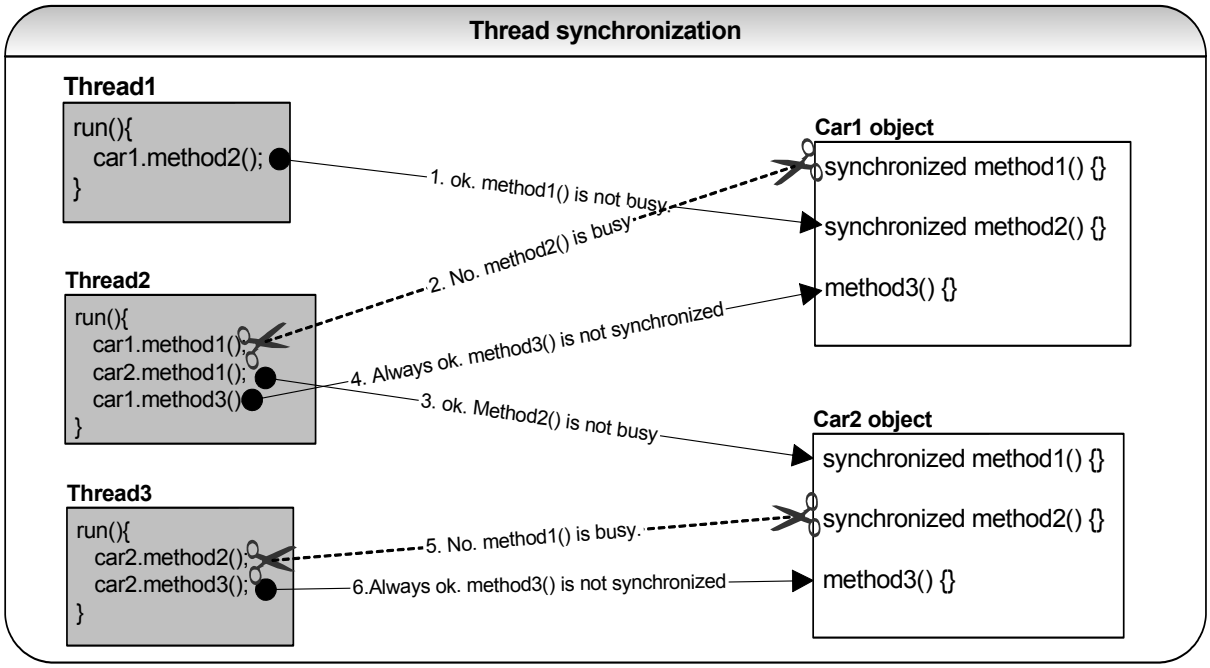
Let's look at a sample code where **count** is a shared resource. The consumer thread will wait inside the consume() method on the producer thread, until the producer thread increments the count inside the produce() method and subsequently notifies the consumer thread. Once it has been notified, the consumer thread waiting inside the consume() method will give up its waiting state and completes its method by consuming the count (i.e. decrementing the count).

Thread communication (Consumer vs Producer threads)

```
Class ConsumerProducer {

  private int count;

  public synchronized void consume(){
     while(count == 0) {
        try{
          wait()
        }
        catch(InterruptedException ie) {
          //keep trying
        }
     }
     count --; //consumed
  }

  private synchronized void produce(){
    count++;
    notify(); // notify the consumer that count has been incremented.
  }
}
```

Q 43: If 2 different threads hit 2 different synchronized methods in an object at the same time will they both continue? LF

A 43: No. Only one method can acquire the lock.



Q 44: Explain threads blocking on I/O? **LF**

A 44: Occasionally threads have to block on conditions other than object locks. I/O is the best example of this. Threads block on I/O (i.e. enters the waiting state) so that other threads may execute while the I/O operation is performed. When threads are blocked (say due to time consuming reads or writes) on an I/O call inside an object's synchronized method and also if the other methods of the object are also synchronized then the object is essentially frozen while the thread is blocked.

Be sure to not synchronize code that makes blocking calls, or make sure that a non-synchronized method exists on an object with synchronized blocking code. Although this technique requires some care to ensure that the resulting code is still thread safe, it allows objects to be responsive to other threads when a thread holding its locks is blocked.

Note: The **java.nio.*** package was introduced in JDK1.4. The coolest addition is nonblocking I/O (aka NIO that stands for New I/O). Refer **Q20** in Java section for NIO.

Note: Q45 & **Q46** are very popular questions on design patterns.

Q 45: What is a **singleton** pattern? How do you code it in Java? **DP** **MI** **CO**

A 45: A singleton is a class that can be instantiated **only one time in a JVM per class loader**. Repeated calls always return the same instance. Ensures that a class has only one instance, and provide a **global point of access**. It can be an issue if singleton class gets loaded by multiple class loaders.

```
public class OnlyOne {

   private static OnlyOne one = new OnlyOne();

   private OnlyOne(){… }   //private constructor. This class cannot be instantiated from outside.

   public static OnlyOne getInstance() {
         return one;
   }
}
```

To use it:

```
//No matter how many times you call, you get the same instance of the object.

OnlyOne myOne = OnlyOne.getInstance();
```

Note: The constructor must be explicitly declared and should have the private access modifier, so that it cannot be instantiated from out side the class. The only way to instantiate an instance of class *OnlyOne* is through the **getInstance()** method with a public access modifier.

When to use: Use it when only a single instance of an object is required in memory for a single point of access. For example the following situations require a **single point of access**, which gets invoked from various parts of the code.

- Accessing application specific properties through a singleton object, which reads them for the first time from a properties file and subsequent accesses are returned from in-memory objects. Also there could be another piece of code, which periodically synchronizes the in-memory properties when the values get modified in the underlying properties file. This piece of code accesses the in-memory objects through the singleton object (i.e. global point of access).

- Accessing in-memory object cache or object pool, or non-memory based resource pools like sockets, connections etc through a singleton object (i.e. global point of access).

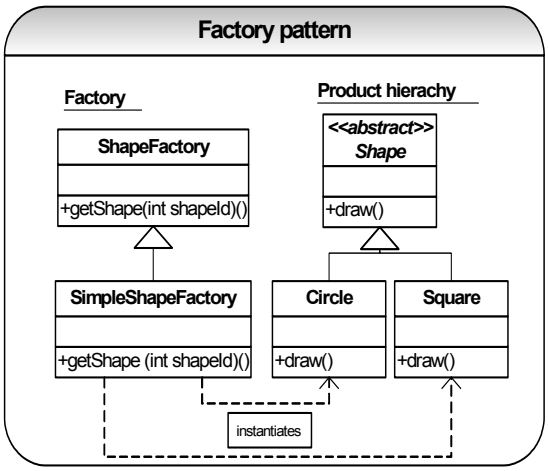
What is the difference between a singleton class and a static class? Static class is one approach to make a class singleton by declaring the class as "final" so that it cannot be extended and declaring all the methods as static so that you can't create any instance of the class and can call the static methods directly.

Q 46: What is a factory pattern? DP CO

A 46: A **Factory method pattern** (aka **Factory pattern**) is a creational pattern. The creational patterns abstract the object instantiation process by hiding how the objects are created and make the system independent of the object creation process. An **Abstract factory** pattern is one level of abstraction higher than a factory method pattern, which means it returns the factory classes.

Factory method pattern (aka Factory pattern)

Factory for what? Factory pattern returns one of the several product subclasses. You should use a factory pattern If you have a super class and a number of sub-classes, and based on some data provided, you have to return the object of one of the subclasses. Let's look at a sample code:




```
public interface Const {
   public static final int SHAPE_CIRCLE =1;
   public static final int SHAPE_SQUARE =2;
   public static final int SHAPE_HEXAGON =3;
}

public class ShapeFactory {
    public abstract Shape getShape(int shapeId);
}

public class SimpleShapeFactory extends
        ShapeFactory  throws BadShapeException  {
```

Abstract factory pattern

An **Abstract factory** pattern is one level of abstraction higher than a factory method pattern, which means the **abstract factory returns the appropriate factory classes**, which will later on return one of the product subclasses. Let's look at a sample code:

```
public class ComplexShapeFactory extends  ShapeFactory {
                throws BadShapeException  {
   public Shape getShape(int shapeTypeId){
      Shape shape = null;
      if(shapeTypeId == Const.SHAPE_HEXAGON) {
            shape = new Hexagon();//complex shape
       }
       else throw new BadShapeException
                  ("shapeTypeId=" +  shapeTypeId);
        return  shape;
    }
}
```

Now let's look at the abstract factory, which returns one of the types of ShapeFactory:

```
public class ShapeFactoryType
          throws   BadShapeFactoryException {

    public static final int TYPE_SIMPLE = 1;
    public static final int TYPE_COMPLEX = 2;

    public ShapeFactory getShapeFactory(int type) {

       ShapeFactory sf = null;

       if(type == TYPE_SIMPLE) {
          sf = new SimpleShapeFactory();
       }
       else if (type == TYPE_COMPLEX) {
          sf = new ComplexShapeFactory();
       }
       else throw new BadShapeFactoryException("No factory!!");
```

```
    public Shape getShape(int shapeTypeId){
        Shape shape = null;
        if(shapeTypeId == Const.SHAPE_CIRCLE) {
            //in future can reuse or cache objects.
            shape = new Circle();
        }
        else if(shapeTypeId == Const.SHAPE_SQUARE) {
            //in future can reuse or cache objects
            shape = new Square();
        }
        else throw new BadShapeException
                (“ShapeTypeId=”+ shapeTypeId);

      return shape;
    }
}
```

Now let’s look at the calling code, which uses the factory:

```
ShapeFactory factory = new SimpleShapeFactory();

//returns a Shape but whether it is a Circle or a
//Square is not known to the caller.
Shape s =  factory.getShape(1);
s.draw(); // circle is drawn

//returns a Shape but whether it is a Circle or a
//Square is not known to the caller.
s = factory.getShape(2);
s.draw(); //Square is drawn
```

```
        return sf;
    }
}
```

Now let’s look at the calling code, which uses the factory:

```
ShapeFactoryType abFac = new  ShapeFactoryType();
ShapeFactory factory = null;
Shape s = null;

//returns a ShapeFactory but whether it is a
//SimpleShapeFactory or a ComplexShapeFactory is not
//known to the caller.
factory = abFac.getShapeFactory(1);//returns SimpleShapeFactory
//returns a Shape but whether it is a Circle or a Pentagon is
//not known to the caller.
s = factory.getShape(2); //returns square.
s.draw(); //draws a square

//returns a ShapeFactory but whether it is a
//SimpleShapeFactory or a ComplexShapeFactory is not
//known to the caller.
factory = abFac.getShapeFactory(2);
//returns a Shape but whether it is a Circle or a Pentagon is
//not known to the caller.
s = factory.getShape(3); //returns a pentagon.
s.draw(); //draws a pentagon
```

Why use factory pattern or abstract factory pattern? Factory pattern returns an instance of several (product hierarchy) subclasses (like ***Circle, Square*** etc), but the calling code is unaware of the actual implementation class. The calling code invokes the method on the interface (for example ***Shape***) and using polymorphism the correct draw() method gets invoked [Refer **Q8** in Java section for polymorphism]. So, as you can see, the factory pattern reduces the coupling or the dependencies between the calling code and called objects like *Circle*, *Square* etc. This is a very powerful and common feature in many frameworks. You do not have to create a new *Circle* or a new *Square* on each invocation as shown in the sample code, which is for the purpose of illustration and simplicity. In future, to conserve memory you can decide to cache objects or reuse objects in your factory with no changes required to your calling code. You can also load objects in your factory based on attribute(s) read from an external properties file or some other condition. Another benefit going for the factory is that unlike calling constructors directly, factory patterns have more meaningful names like getShape(…), getInstance(…) etc, which may make calling code more clear.

Can we use the singleton pattern within our factory pattern code? Yes. Another important aspect to consider when writing your factory class is that, it does not make sense to create a new factory object for each invocation as it is shown in the sample code, which is just fine for the illustration purpose.

```
ShapeFactory factory = new SimpleShapeFactory();
```

To overcome this, you can incorporate the singleton design pattern into your factory pattern code. The singleton design pattern will create only a single instance of your *SimpleShapeFactory* class. Since an abstract factory pattern is unlike factory pattern, where you need to have an instance for each of the two factories (i.e. *SimpleShapeFactory* and ComplexShapeFactory) returned, you can still incorporate the singleton pattern as an access point and have an instance of a *HashMap,* store your instances of both factories. Now your calling method uses a static method to get the same instance of your factory, hence conserving memory and promoting object reuse:

```
ShapeFactory factory = ShapeFactory. Ge/tFactoryInstance();
factory.getShape();
```

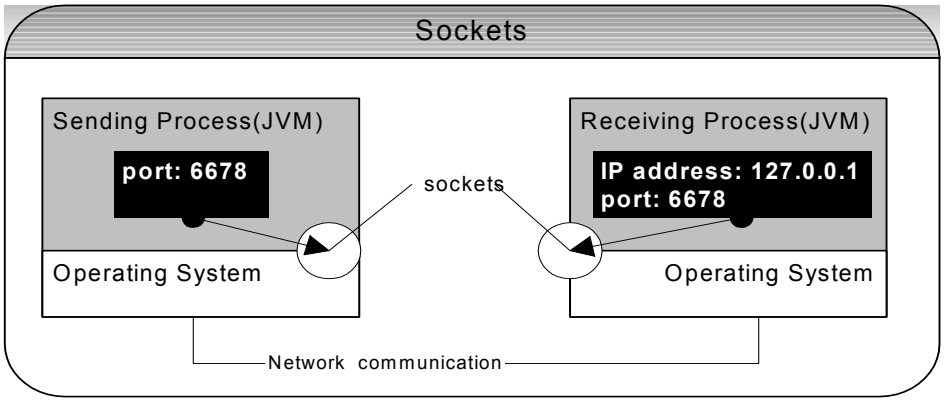
Note: Since questions on singleton pattern and factory pattern are commonly asked in the interviews, they are included as part of this section. To learn more about design patterns refer **Q11** in How would you go about section…?

Q 47: What is a socket? How do you facilitate inter process communication in Java? LF

A 47: A socket is a communication channel, which facilitates **inter-process communication** (For example communicating between two JVMs, which may or may not be running on two different physical machines)**.** A socket is an endpoint for communication. There are two kinds of sockets, depending on whether one wishes to use a connectionless or a connection-oriented protocol. The connectionless communication protocol of the Internet is called UDP. The connection-oriented communication protocol of the Internet is called TCP. UDP sockets are also called datagram sockets. Each socket is uniquely identified on the entire Internet with two numbers. The first number is a 128-bit integer called the Internet Address (or **IP address**). The second number is a 16-bit integer called the **port** of the socket. The IP address is the location of the machine, which you are trying to connect to and the port number is the port on which the server you are trying to connect is running. The port numbers 0 to 1023 are reserved for standard services such as e-mail, FTP, HTTP etc.

The lifetime of the socket is made of 3 phases: **Open Socket → Read and Write to Socket → Close Socket**

To make a socket connection you need to know two things: An IP address and port on which to listen/connect. In Java you can use the ***Socket*** (client side*)* and ***ServerSocket*** *(*Server side*)* classes.



Q 48: How will you call a Web server from a stand alone Java application? LF

A 48: Using the **java.net.URLConnection** and its subclasses like HttpURLConnection and JarURLConnection.

URLConnection	HttpClient (browser)
Supports HEAD, GET, POST, PUT, DELETE, TRACE and OPTIONS	Supports HEAD, GET, POST, PUT, DELETE, TRACE and OPTIONS.
Does not support cookies.	Does support cookies.
Can handle protocols other than http like ftp, gopher, mailto and file.	Handles only http.

Java – Swing

Q 49: What is the difference between AWT and Swing? LF DC

A 49: Swing provides a richer set of components than AWT. They are 100% Java-based. There are a few other advantages to Swing over AWT:

- Swing provides both additional components like JTable, JTree etc and added functionality to AWT-replacement components.
- Swing components can change their appearance based on the current "look and feel" library that's being used.
- Swing components follow the **Model-View-Controller** (MVC) paradigm, and thus can provide a much more flexible UI.
- Swing provides "extras" for components, such as: icons on many components, decorative borders for components, tool tips for components etc.
- Swing components are lightweight (less resource intensive than AWT).

- Swing provides built-in **double buffering** (which means an off-screen buffer [image] is used during drawing and then the resulting bits are copied onto the screen. The resulting image is smoother, less flicker and quicker than drawing directly on the screen).
- Swing provides paint debugging support for when you build your own component i.e.-slow motion rendering.

Swing also has a few disadvantages:

- If you're not very careful when programming, it can be slower than AWT (all components are drawn).
- Swing components that look like native components might not behave exactly like native components.

Q 50: Explain the Swing *Action* architecture? LF DP

A 50: The Swing *Action* architecture is used to implement shared behaviour between two or more user interface components. For example, the menu items and the tool bar buttons will be performing the same action no matter which one is clicked. Another distinct advantage of using actions is that when an action is disabled then all the components, which use the Action, become disabled.

Design pattern: The *javax.swing.Action* interface extends the *ActionListener* interface and is an abstraction of a command that does not have an explicit UI component bound to it. The Action architecture is an implementation of a **command design pattern**. This is a powerful design pattern because it allows the separation of controller logic of an application from its visual representation. This allows the application to be easily configured to use different UI elements without having to re-write the control or call-back logic.

Defining action classes:

```
class FileAction extends AbstractAction {
   //Constructor
   FileAction(String name) {
        super(name);
   }

   public void actionPerformed(ActionEvent ae){
       //add action logic here
   }
}
```

To add an action to a menu bar:

```
JMenu fileMenu =  new JMenu("File");
FileAction newAction = new FileAction("New");
JMenuItem item =  fileMenu.add(newAction);
item.setAccelaror(KeyStroke.getKeyStroke('N', Event.CTRL_MASK));
```

To add action to a toolbar

```
private JToolBar toolbar = new JToolBar();
toolbar.add(newAction);
```

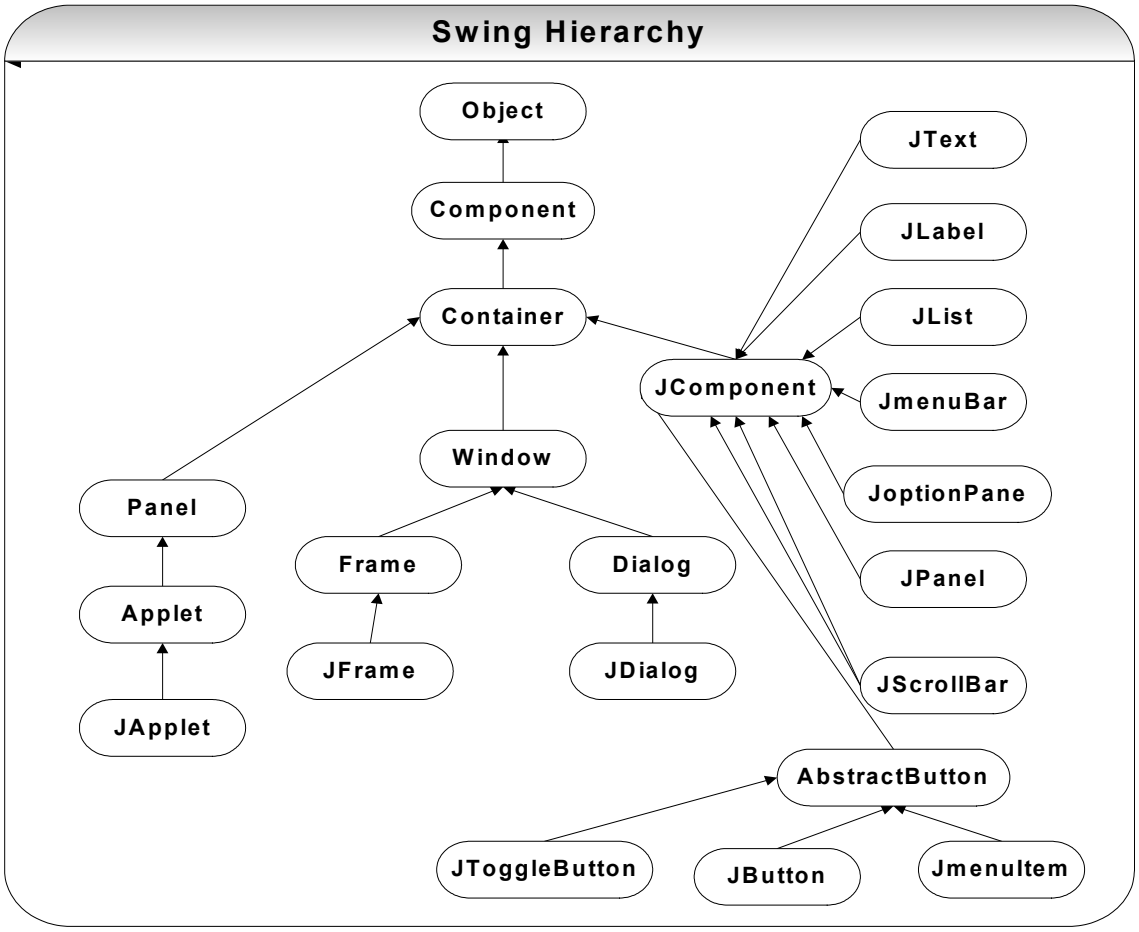
So, an *action* object is a listener as well as an action.

Q 51: If you add a component to the CENTER of a border layout, which directions will the component stretch? LF

A 51: The component will stretch both horizontally and vertically. It will occupy the whole space in the middle.

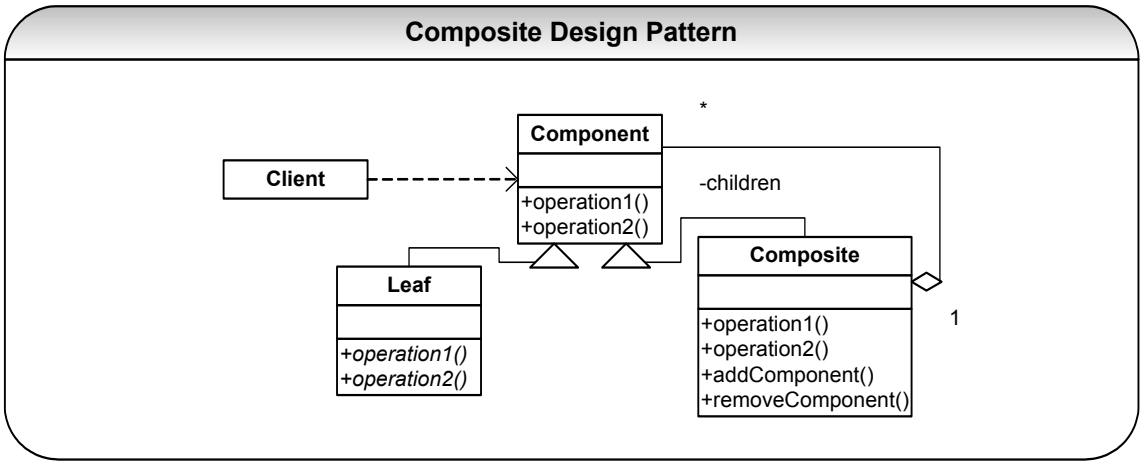
Q 52: What is the base class for all Swing components? LF

A 52: All the Swing components start with 'J'. The hierarchy diagram is shown below. **JComponent** is the base class.



(**Diagram source**: http://www.particle.kth.se/~fmi/kurs/PhysicsSimulation/Lectures/07A/swingDesign.html)

Design pattern: As you can see from the above diagram, containers collect components. Sometimes you want to add a container to another container. So, a container should be a component. For example container.getPreferredSize() invokes getPreferredSize() of all contained components. **Composite design pattern** is used in GUI components to achieve this. A composite object is an object, which contains other objects. Composite design pattern manipulates composite objects just like you manipulate individual components. Refer **Q11** in How would you go about…? section.



Q 53: Explain the Swing *event dispatcher* mechanism? LF CI PI

A 53: Swing components can be accessed by the Swing **event dispatching thread**. A few operations are guaranteed to be thread-safe but most are not. Generally the Swing components should be accessed through this ***event-***

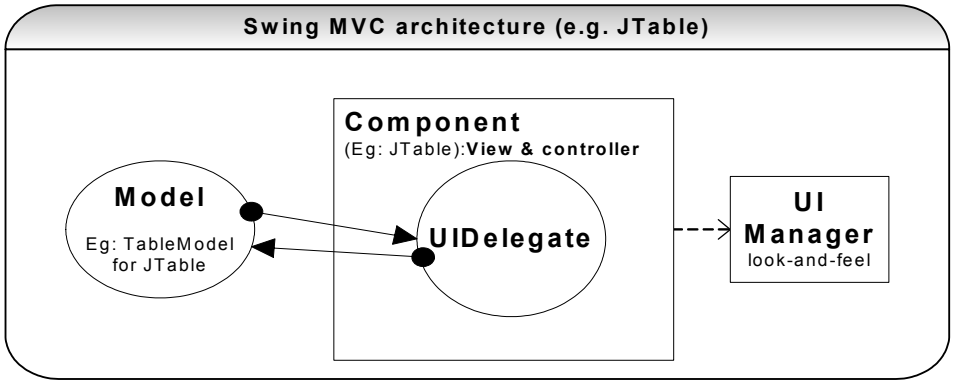
dispatching thread. The *event-dispatching* thread is a thread that executes drawing of components and event-handling code. For example the paint() and actionPerformed() methods are automatically executed in the *event-dispatching thread.* Another way to execute code in the *event-dispatching thread* from outside event-handling or drawing code, is using *SwingUtilities* **invokeLater()** or **invokeAndWait()** method. **Swing lengthy initialization tasks (e.g. I/O bound and computationally expensive tasks), should not occur in the *event-dispatching thread* because this will hold up the dispatcher thread**. If you need to create a new thread for example, to handle a job that's computationally expensive or I/O bound then you can use the thread utility classes such as *SwingWorker* or *Timer* without locking up the *event-dispatching thread.*

- ***SwingWorker*** – creates a background thread to execute time consuming operations.
- ***Timer*** – creates a thread that executes at certain intervals.

However after the lengthy initialization the GUI update should occur in the event dispatching thread, for thread safety reasons. We can use **invokeLater()** to execute the GUI update in the *event-dispatching thread.* The other scenario where **invokeLater()** will be useful is that the GUI must be updated as a result of non-AWT event.

Q 54: What do you understand by MVC as used in a JTable? LF DP

A 54: MVC stands for **M**odel **V**iew **C**ontroller architecture. Swing "J" components (e.g. JTable, JList, JTree etc) use a **modified version of MVC**. MVC separates a model (or data source) from a presentation and the logic that manages it.



- **Component** (e.g. *JTable, JTree, and JList*): coordinates actions of model and the UI delegate. Each generic component class handles its own individual ***view-and-controller*** responsibilities.
- **Model** (e.g. *TableModel*): charged with storing the data.
- **UIDelegate**: responsible for getting the data from model and rendering it to screen. It delegates any look-and-feel aspect of the component to the **UI Manager**.

Q 55: Explain layout managers? LF

A 55: Layout managers are used for arranging GUI components in windows. The standard layout managers are:

- ***FlowLayout*****:** Default layout for **Applet** and **Panel.** Lays out components from left to right, starting new rows if necessary.
- ***BorderLayout*****:** Default layout for **Frame** and **Dialog.** Lays out components in north, south, east, west and center. All extra space is placed on the center.
- ***CardLayout:*** stack of same size components arranged inside each other. Only one is visible at any time. Used in TABs.
- ***GridLayout:*** Makes a bunch of components equal in size and displays them in the requested number of rows and columns.
- ***GridBagLayout*****:** Most complicated but the most flexible. It aligns components by placing them within a grid of cells, allowing some components to span more than one cell. The rows in the grid aren't necessarily all the same height, similarly, grid columns can have different widths as well.

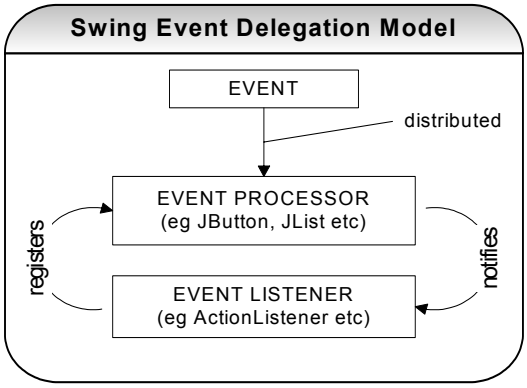
- ***BoxLayout:*** is a full-featured version of FlowLayout. It stacks the components on top of each other or places them in a row.

Complex layouts can be simplified by using nested containers for example having *panels* within *panels* and each *panel* can use its own *LayoutManager*. It is also possible to write your own layout manager or use manual positioning of the GUI components. **Note:** Further reading on each *LayoutManagers* is recommended for Swing developers.

Design pattern: The AWT containers like panels, dialog boxes, windows etc do not perform the actual laying out of the components. They delegate the layout functionality to layout managers. The layout managers make use of the **strategy design pattern**, which encapsulates family of algorithms for laying out components in the containers. If a particular layout algorithm is required other than the default algorithm, an appropriate layout manager can be instantiated and plugged into the container (e.g. panels by default uses the FlowLayout but it can be changed by executing → panel.setLayout(new GridLayout(4,5))). This enables the layout algorithms to vary independently from the containers that uses them, this is one of the key benefits of the strategy pattern.

Q 56: Explain the Swing delegation event model? **LF**
A 56: In this model, the objects that receive user events notify the registered listeners of the user activity. In most cases the event receiver is a component.

- **Event Types:** ActionEvent, KeyEvent, MouseEvent, WindowEvent etc.
- **Event Processors:** JButton, JList etc.
- **EventListeners:** ActionListener, ComponentListener, KeyListener etc.



Java – Applet

Q 57: How will you initialize an applet? **LF**
A 57: By writing your initialization code in the applet's **init()** method or applet's **constructor**.

Q 58: What is the order of method invocation in an applet? **LF**
A 58: The Applet's life cycle methods are as follows:

- ***public void init()*** **:** Initialization method called only once by the browser.

- ***public void start()*** **:** Method called after init() and contains code to start processing. If the user leaves the page and returns without killing the current browser session, the start () method is called without being preceded by init ().

- ***public void stop()*** **:** Stops all processing started by start (). Done if user moves off page.

- ***public void destroy()*** **:** Called if current browser session is being terminated. Frees all resources used by the applet.

Q 59: How would you communicate between applets and servlets? LF

A 59: We can use the **java.net.URLConnection** and **java.net.URL** classes to open a standard HTTP connection and "**tunnel**" to a Web server. The server then passes this information to the servlet. Basically, the applet pretends to be a Web browser, and the servlet doesn't know the difference. As far as the servlet is concerned, the applet is just another HTTP client. Applets can communicate with servlets using GET or POST methods.

The parameters can be passed between the applet and the servlet as **name value pairs.**

```
http://www.foo.com/servlet/TestServlet?LastName=Jones&FirstName=Joe).
```

Objects can also be passed between applet and servlet using object serialization. Objects are serialized to and from the inputstream and outputstream of the connection respectively.

Q 60: How will you communicate between two Applets? LF

A 60: All the applets on a given page share the same AppletContext. We obtain this applet context as follows:

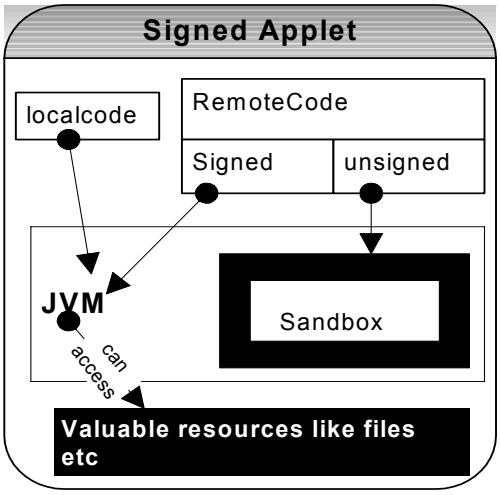
```
AppletContext ac = getAppletContext();
```

AppletContext provides applets with methods such as getApplet(name), getApplets(),getAudioClip, getImage, showDocument and showStatus().

Q 61: What is a signed Applet? LF SE

A 61: A signed Applet is a **trusted** Applet. By default, and for security reasons, Java applets are contained within a "**sandbox**". Refer to the diagram below:

This means that the applets can't do anything, which might be construed as threatening to the user's machine (e.g. reading, writing or deleting local files, putting up message windows, or querying various system parameters). Early browsers had no provisions for Java applets to reach outside of the sandbox. Recent browsers, however (Internet Explorer 4 on Windows etc), have provisions to give "**trusted**" applets the ability to work outside the sandbox. For this power to be granted to one of your applets, the applet's code must be digitally signed with your unforgeable digital ID, and then the user must state that he trusts applets signed with your ID. The untrusted applet can request to have privileges outside the sand box but will have to request the user for privileges every time it executes. But with the trusted applet the user can choose to remember their answer to the request, which means they won't be asked again.



Q 62: What is the difference between an applet and an application? LF

A 62:

Applet	Application
Applets don't have a main method. They operate on life cycle methods init(), start(), stop(), destroy() etc.	Has a static main() method.
Applets can be embedded in HTML pages and	Has no support for embedding or downloading. Has

downloaded over the Internet. Has a sand box security model.	no inherent security restriction.
Can only be executed within a Java compatible container like browser, appletviewer etc.	Applications are executed at command line by java.exe.

Java – Performance and Memory leaks

Q 63: How would you improve performance of a Java application? **PI** **BP**

A 63:

- **Pool valuable system resources** like threads, database connections, socket connections etc. Emphasise on reuse of threads from a pool of threads. Creating new threads and discarding them after use can adversely affect performance. Also consider using multi-threading in your single-threaded applications where possible to enhance performance. Optimze the pool sizes based on system and application specifications and requirements.

- **Optimize your I/O operations:** use buffering (Refer **Q21** in Java section) when writing to and reading from files and/or streams. Avoid writers/readers if you are dealing with only ASCII characters. You can use streams instead, which are faster. Avoid premature flushing of buffers. Also make use of the performance and scalability enhancing features such as non-blocking and asynchronous I/O, mapping of file to memory etc offered by the NIO (New I/O).

- **Minimize network overheads** by retrieving several related items simultaneously in one remote invocation if possible. Remote method invocations involve a network round-trip, marshalling and unmarshalling of parameters, which can cause huge performance problems if the remote interface is poorly designed. (Refer **Q125** in Enterprise section).

- **Establish whether you have a potential memory problem and manage your objects efficiently**: remove references to the short-lived objects from long-lived objects like Java collections etc (Refer **Q64** in Java section) to minimise any potential memory leaks. Also reuse objects where possible. It is cheaper to recycle objects than creating new objects each time. Avoid creating extra objects unnecessarily. For example use mutable *StringBuffer/StringBuilder* classes instead of immutable *String* objects in computation expensive loops as discussed in **Q17** in Java section. Automatic garbage collection is one of the most highly touted conveniences of Java. However, it comes at a price. Creating and destroying objects occupies a significant chunk of the JVM's time. Wherever possible, you should look for ways to minimise the number of objects created in your code:

  - If repeating code within a loop, avoid creating new objects for each iteration. Create objects before entering the loop (i.e. outside the loop) and reuse them if possible.

  - For complex objects that are used frequently, consider creating a pool of recyclable objects rather than always instantiating new objects. This adds additional burden on the programmer to manage the pool, but in select cases can represent an order of magnitude performance gain.

  - Use lazy initialization when you want to distribute the load of creating large amounts of objects. Use lazy initialization only when there is merit in the design.

- **Where applicable apply the following performance tips in your code**:

  - Use ArrayLists, HashMap etc as opposed to Vector, Hashtable etc where possible. This is because the methods in ArrayList, HashMap etc are not synchronized (Refer **Q13** in Java Section). Even better is to use just arrays where possible.

  - Set the initial capacity of a collection (e.g. *ArrayList*, *HashMap* etc) and *StringBuffer/StringBuilder* appropriately. This is because these classes must grow periodically to accommodate new elements. So, if you have a very large *ArrayList* or a *StringBuffer*, and you know the size in advance then you can speed things up by setting the initial size appropriately. (Refer **Q15, Q17** in Java Section).

- Minimise the use of **casting** or runtime type checking like ***instanceof*** in frequently executed methods or in loops. The "casting" and "instanceof" checks for a class marked as final will be faster. Using "instanceof" construct is not only ugly but also unmaintainable. Look at using **visitor pattern** (Refer **Q11** in How would you go about…? section) to avoid "instanceof" construct.

- Do not compute constants inside a large loop. Compute them outside the loop. For applets compute it in the init() method.

- Exception creation can be expensive because it has to create the full stack trace. The stack trace is obviously useful if you are planning to log or display the exception to the user. But if you are using your exception to just control the flow, which is not recommended, then throw an exception, which is pre-created. An efficient way to do this is to declare a public static final *Exception* in your exception class itself.

- Avoid using System.out.println and use logging frameworks like Log4J etc, which uses I/O buffers (Refer **Q21** in Java section).

- Minimise calls to Date, Calendar, etc related classes.

- Minimise JNI calls in your code.

Note: Set performance requirements in the specifications, include a performance focus in the analysis and design and also create a performance test environment.

Q 64: How would you detect and minimise memory leaks in Java? **MI** **BP**

A 64: In Java memory leaks are caused by poor program design where object references are long lived and the garbage collector is unable to reclaim those objects.

Detecting memory leaks:

- Use tools like JProbe, OptimizeIt etc to detect memory leaks.

- Use operating system process monitors like task manager on NT systems, ps, vmstat, iostat, netstat etc on UNIX systems.

- Write your own utility class with the help of totalMemory() and freeMemory() methods in the Java *Runtime* class. Place these calls in your code strategically for pre and post memory recording where you suspect to be causing memory leaks. An even better approach than a utility class is using **dynamic proxies** (Refer **Q11** in How would you go about section…) or **Aspect Oriented Programming** (**AOP**) for pre and post memory recording where you have the control of activating memory measurement only when needed. (Refer **Q3 – Q5** in Emerging Technologies/Frameworks section).

Minimising memory leaks:

In Java, typically memory leak occurs when **an object of a longer lifecycle has a reference to objects of a short life cycle**. This prevents the objects with short life cycle being garbage collected. The developer must remember to remove the references to the short-lived objects from the long-lived objects. Objects with the same life cycle do not cause any issues because the garbage collector is smart enough to deal with the circular references (Refer **Q33** in Java section).

- Design applications with an object's life cycle in mind, instead of relying on the clever features of the JVM. Letting go of the object's reference in one's own class as soon as possible can mitigate memory problems. **Example:** myRef = null;

- Unreachable collection objects can magnify a memory leak problem. In Java it is easy to let go of an entire collection by setting the root of the collection to null. The garbage collector will reclaim all the objects (unless some objects are needed elsewhere).

- Use weak references (Refer **Q32** in Java section) if you are the only one using it. The **WeakHashMap** is a combination of *HashMap* and *WeakReference*. This class can be used for programming problems where you need to have a *HashMap* of information, but you would like that information to be garbage collected if you are the only one referencing it.

- Free native system resources like AWT frame, files, JNI etc when finished with them. **Example:** *Frame*, *Dialog*, and *Graphics* classes require that the method dispose() be called on them when they are no longer used, to free up the system resources they reserve.

Q 65: Why does the JVM crash with a core dump or a Dr.Watson error? **MI**

A 65: Any problem in pure Java code throws a Java exception or error. Java exceptions or errors will not cause a core dump (on UNIX systems) or a Dr.Watson error (on WIN32systems). Any serious Java problem will result in an ***OutOfMemoryError*** thrown by the JVM with the stack trace and consequently JVM will exit. These Java stack traces are very useful for identifying the cause for an abnormal exit of the JVM. So is there a way to know that ***OutOfMemoryError*** is about to occur? The Java JDK 1.5 has a package called java.lang.management which has useful JMX beans that we can use to manage the JVM. One of these beans is the MemoryMXBean.

An ***OutOfMemoryError*** can be thrown due to one of the following 4 reasons:

- JVM may have a memory leak due to a bug in its internal heap management implementation. But this is highly unlikely because JVMs are well tested for this.

- The application may not have enough heap memory allocated for its running. You can allocate more JVM heap size (with –Xmx parameter to the JVM) or decrease the amount of memory your application takes to overcome this. To increase the heap space:

```
Java -Xms1024M -Xmx1024M
```

  Care should be taken not to make the –Xmx value too large because it can slow down your application. The secret is to make the maximum heap size value the right size.

- Another not so prevalent cause is the running out of a memory area called the "**perm**" which sits next to the heap. All the binary code of currently running classes is archived in the "perm" area. The 'perm' area is important if your application or any of the third party jar files you use dynamically generate classes. **For example:** "perm" space is consumed when XSLT templates are dynamically compiled into classes, J2EE application servers, JasperReports, JAXB etc use Java reflection to dynamically generate classes and/or large amount of classes in your application. To increase perm space:

```
Java  -XX:PermSize=256M -XX:MaxPermSize=256M
```

- The fourth and the most common reason is that you may have a memory leak in your application as discussed in **Q64** in Java section.

[Good read/reference: "**Know Your Worst Friend, the Garbage Collector**" http://java.sys-con.com/read/84695.htm by Romain Guy]

So why does the JVM crash with a core dump or Dr.Watson error?

Both the core dump on UNIX operating system and Dr.Watson error on WIN32 systems mean the same thing. The JVM is a process like any other and when a process crashes a core dump is created. A core dump is a memory map of a running process. This can happen due to one of the following reasons:

- Using JNI (Java Native Interface) code, which has a fatal bug in its native code. **Example:** using Oracle OCI drivers, which are written partially in native code or jdbc-odbc bridge drivers, which are written in non Java code. Using 100% pure Java drivers (communicates directly with the database instead of through client software utilizing the JNI) instead of native drivers can solve this problem. We can use Oracle thin driver, which is a 100% pure Java driver.

- The operating system on which your JVM is running might require a patch or a service pack.

- The JVM implementation you are using may have a bug in translating system resources like threads, file handles, sockets etc from the platform neutral Java byte code into platform specific operations. If this JVM's translated native code performs an illegal operation then the **operating system will instantly kill the process and mostly will generate a core dump file,** which is a hexadecimal file indicating program's state in memory at the time of error. The core dump files are generated by the operating system in response to certain signals. Operating system signals are responsible for notifying certain events to its threads and processes. The JVM can also intercept certain signals like **SIGQUIT** which is kill -3 < process id > from the operating system and it responds to this signal by printing out a Java stack trace and then continue to run.

The JVM continues to run because the JVM has a special built-in debug routine, which will trap **the signal -3**. On the other hand signals like **SIGSTOP** (kill -23 <process id>) and **SIGKILL** (kill -9 <process id>) will cause the JVM process to stop or die. The following JVM argument will indicate JVM not to pause on **SIGQUIT** signal from the operating system.

```
Java –Xsqnopause
```

Java – Personal

Q 66: Did you have to use any design patterns in your Java project? **DP**

A 66: Yes. Refer **Q10 [Strategy], Q14 [Iterator], Q20 [Decorator], Q31 [Visitor], Q45 [Singleton], Q46 [Factory], Q50 [Command],** and **Q54 [MVC]** in Java section and **Q11** in How would you go about… section. **Note:** Learning of other patterns recommended (Gang of Four Design Patterns).

Resource: http://www.patterndepot.com/put/8/JavaPatterns.htm.

Why use design patterns, you may ask (Refer **Q5** in Enterprise section). Design patterns are worthy of mention in your CV and interview. Design patterns have a number of advantages:

- Capture design experience from the past.
- Promote reuse without having to reinvent the wheel.
- Define the system structure better.
- Provide a common design vocabulary.

Some advice if you are just starting on your design pattern journey:

- If you are not familiar with UML, now is the time. UML is commonly used to describe patterns in pattern catalogues, including class diagrams, sequence diagrams etc. (Refer **Q106 - Q109** in Enterprise section).

- When using patterns, it is important to define a naming convention. It will be much easier to manage a project as it grows to identify exactly what role an object plays with the help of a naming convention e.g. AccountFacility**BusinessDelegate**, AccountFacility**Factory**, AccountFacility**ValueObject,** Account**Decorator**, Account**Visitor,** Account**TransferObject** (or AccountFacility**VO** or Account**TO**).

- Make a list of requirements that you will be addressing and then try to identify relevant patterns that are applicable.

Q 67: Tell me about yourself or about some of the recent projects you have worked with? What do you consider your most significant achievement? Why do you think you are qualified for this position? Why should we hire you and what kind of contributions will you make?

A 67: **[Hint:]** Pick your recent projects and brief on it. Also is imperative that during your briefing, you demonstrate how you applied your skills and knowledge in some of the following areas:

- Design concepts and design patterns: **How you understand and applied them**.
- Performance and memory issues: **How you identified and fixed them**.
- Exception handling and best practices: **How you understand and applied them.**
- Multi-threading and concurrent access: **How you identified and fixed them**.

Some of the questions in this section can help you prepare your answers by relating them to your current or past work experience. For example:

- **Design Concepts**: Refer **Q5, Q6, Q7, Q8, Q9** etc
- **Design Patterns**: Refer **Q10, Q14, Q20, Q31, Q45, Q46, Q50** etc [Refer **Q11** in How would you go about…? section]
- **Performance issues**: Refer **Q21, Q63** etc
- **Memory issues**: Refer **Q32, Q64, Q65** etc
- **Exception Handling**: Refer **Q34, Q35** etc
- **Multi-threading (Concurrency issues)**: Refer **Q29, Q40** etc

Demonstrating your knowledge in the above mentioned areas will improve your chances of being successful in your Java/J2EE interviews. 90% of the interview questions are asked based on your own resume. So in my view it is also very beneficial to mention how you demonstrated your knowledge/skills by stepping through a recent project on your resume.

The two other areas, which I have not mentioned in this section, which are also very vital, are transactions and security. These two areas will be covered in the next section, which is the Enterprise section (J2EE, JDBC, EJB, JMS, SQL, XML etc).

Even if you have not applied these skills knowingly or you have not applied them at all, just demonstrating that you have the knowledge and an appreciation will help you improve your chances in the interviews. Also mention any long hours worked to meet the deadline, working under pressure, fixing important issues like performance issues, running out of memory issues etc**.**

Q 68: Why are you leaving your current position?
A 68: [Hint]

- Do not criticize your previous employer or coworkers or sound too opportunistic.
- It is fine to mention a major problem like a buy out, budget constraints, merger or liquidation.
- You may also say that your chance to make a contribution is very low due to company wide changes or looking for a more challenging senior or designer role.

Q 69: What do you like and/or dislike most about your current and/or last position?
A 69: [Hint]

The interviewer is trying to find the compatibility with the open position. So

Do not say anything like:

- You dislike overtime.
- You dislike management or coworkers etc.

It is **safe to say**:

- You like challenges.
- Opportunity to grow into design, architecture, performance tuning etc.
- You dislike frustrating situations like identifying a memory leak problem or a complex transactional or a concurrency issue. You want to get on top of it as soon as possible.

Q 70: How do you handle pressure? Do you like or dislike these situations?
A 70: [Hint]

These questions could mean that the open position is pressure-packed and may be out of control. Know what you are getting into. If you do perform well under stress then give a descriptive example. High achievers tend to perform well in pressure situations.

Q 71: What are your strengths and weaknesses? Can you describe a situation where you took initiative? Can you describe a situation where you applied your problem solving skills?
A 71: [Hint]

Strengths:

- **Taking initiatives** and **being pro-active:** You can illustrate how you took initiative to fix a transactional issue, a performance problem or a memory leak problem.

- **Design skills:** You can illustrate how you designed a particular application using OO concepts.

- **Problem solving skills:** Explain how you will break a complex problem into more manageable sub-sections and then apply brain storming and analytical skills to solve the complex problem. Illustrate how you went about identifying a scalability issue or a memory leak problem.

- **Communication skills:** Illustrate that you can communicate effectively with all the team members, business analysts, users, testers, stake holders etc.
- **Ability to work in a team environment as well as independently:** Illustrate that you are technically sound to work independently as well as have the interpersonal skills to fit into any team environment.
- **Hard working, honest,** and **conscientious etc** are the adjectives to describe you**.**

Weaknesses:

Select a trait and come up with a solution to overcome your weakness. Stay away from personal qualities and concentrate more on professional traits for example:

- I pride myself on being an attention to detail guy but sometimes miss small details. So I am working on applying the 80/20 principle to manage time and details. Spend 80% of my effort and time on 20% of the tasks, which are critical and important to the task at hand.
- Some times when there is a technical issue or a problem I tend to work continuously until I fix it without having a break. But what I have noticed and am trying to practise is that taking a break away from the problem and thinking outside the square will assist you in identifying the root cause of the problem sooner.

Q 72: What are your career goals? Where do you see yourself in 5-10 years?
A 72: [Hint] Be realistic. For example

- Next 2-3 years to become a senior developer or a team lead.
- Next 3-5 years to become a solution designer or an architect.

Note: For **Q66 – Q72 tailor your answers to the job**. Also be prepared for questions like:

- What was the last Java related book or article you read? [Hint]
  - **Mastering EJB** by Ed Roman.
  - **EJB design patterns** by Floyd Marinescu.
  - **Bitter Java** by Bruce Tate.
  - **Thinking in Java** by Bruce Eckel.
- Which Java related website(s) do you use to keep your knowledge up to date? [Hint]
  - http://www.theserverside.com
  - http://www.javaworld.com
  - http://www-136.ibm.com/developerworks/Java
  - http://www.precisejava.com
  - http://www.allapplabs.com
  - http://java.sun.com
  - http://www.martinfowler.com
  - http://www.ambysoft.com
- What past accomplishments gave you satisfaction? What makes you want to work hard? [Hint]
  - Material rewards such as salary, perks, benefits etc naturally come into play but focus on your achievements or accomplishments than on rewards.
- Do you have any role models in software development? [Hint]
  - Scott W. Ambler, Martin Fowler, Ed Roman, Floyd Marinescu, Grady Booch etc.
- Why do you want to work for us? (Research the company prior to the interview).

Java – Key Points

- Java is an object oriented (OO) language, which has built in support for multi-threading, socket communication, automatic memory management (i.e. garbage collection) and also has better portability than other languages across operating systems.
- Java class loaders are **hierarchical** and use a **delegation model**. The classes loaded by a child class loader have **visibility** into classes loaded by its parents up the hierarchy but the reverse is not true.
- Java does not support **multiple implementation inheritance** but supports **multiple interface inheritance**.
- **Polymorphism**, **inheritance** and **encapsulation** are the 3 pillar of an object-oriented language.
- Code reuse can be achieved through either **inheritance** ("is a" relationship) or **object composition** ("has a" relationship). Favour object composition over inheritance.
- When using **implementation inheritance**, make sure that the **subclasses depend only on the behaviour of the superclass**, not the actual implementation. An **abstract** base class usually provides an implementation inheritance.
- Favour **interface inheritance** to **implementation inheritance** because it promotes the deign concept of **coding to interface** and **reduces coupling**. The interface inheritance can achieve code reuse through **object composition**.
- Design by contract specifies the obligations of a calling-method and called-method to each other using **pre-conditions**, **post-conditions** and **class invariants**.
- When using Java collection API, prefer using *ArrayList* or *HashMap* as opposed to *Vector* or *Hashtable* to **avoid any synchronization overhead**. The *ArrayList* or *HashMap* can be externally synchronized for concurrent access by multiple threads.
- Set the initial capacity of a collection appropriately and program in terms of interfaces as opposed to implementations.
- When providing a user defined key class for storing objects in *HashMap*, you should override **equals()**, and **hashCode()** methods from the *Object* class.
- *String* class is immutable and *StringBuffer* and *StringBuilder* classes are mutable. So it is more efficient to use a *StringBuffer* or a *StringBuilder* as opposed to a *String* in a computation intensive situations (ie. in for, while loops).
- **Serialization** is a process of writing an object to a file or a stream. **Transient** variables cannot be serialized.
- Java I/O performance can be improved by using buffering, minimising access to the underlying hard disk and operating systems. Use the NIO package for performance enhancing features like non-blocking I/O operation, buffers to hold data, and memory mapping of files.
- Each time an object is created in Java it goes into the area of memory known as **heap**. The primitive variables are allocated in the **stack** if they are local method variables and in the **heap** if they are class member variables.
- Threads **share the heap spaces** so it is **not thread-safe** and the threads have **their own stack space**, which is **thread-safe**.
- The **garbage collection cannot be forced**, but you can nicely ask the garbage collector to collect garbage.
- There two types of exceptions **checked** (ie compiler checked) and **unchecked** (Runtime Exceptions). It is not advisable to catch type *Exception*.
- A **process** is an execution of a program (e.g. JVM process) but a **thread** is a single execution sequence within the process.
- Threads can be created in Java by either extending the *Thread* class or implementing the *Runnable* interface.

- In Java each object has a lock and a thread can acquire a lock by using the **synchronized** key word. The synchronization key word can be applied in **method level** (coarse-grained lock) or **block level** (fine-grained lock which offers better performance) of code.

- Threads can communicate with each other using **wait(), notify(), and notifyAll()** methods. This communication solves the **consumer-producer** problem.

- Sockets are communication channels, which facilitate inter-process communication.

- Swing uses the **MVC paradigm** to provide loose coupling and action **architecture** to implement a shared behaviour between two or more user interface components.

- Swing components should be accessed through an **event-dispatching thread**. There is a way to access the Swing event-dispatching thread from outside event-handling or drawing code, is using *SwingUtilities'* **invokeLater()** and **invokeAndWait()** methods.

- A signed applet can become a trusted applet, which can work outside the sandbox.

- In Java typically memory leak occurs when an object of longer life cycle has a reference to objects of a short life cycle.

- You can improve performance in Java by :

    1. Pooling your valuable resources like threads, database and socket connections.
    2. Optimizing your I/O operations.
    3. Minimising network overheads, calls to *Date*, *Calendar* related classes, use of "**casting**" or runtime type checking like "**instanceof"** in frequently executed methods/loops, JNI calls, etc
    4. Managing your objects efficiently by caching or recycling them without having to rely on garbage collection.
    5. Using a *StringBuffer* as opposed to *String* and *ArrayList* or *HashMap* as oppose to *Vector* or *Hashtable*
    6. Applying multi-threading where applicable.
    7. Minimise any potential memory leaks.

- Finally, very briefly familiarise yourself with some of the key **design patterns** like:

    1. **Decorator design pattern**: used by Java I/O API. A popular design pattern.
    2. **Reactor design pattern/Observer design pattern**: used by Java NIO API.
    3. **Visitor design pattern**: to avoid instanceof and typecast constructs.
    4. **Factory method/abstract factory design pattern**: popular pattern, which gets frequently asked in interviews.
    5. **Singleton pattern**: popular pattern, which gets frequently asked in interviews.
    6. **Composite design pattern**: used by GUI components and also a popular design pattern
    7. **MVC design pattern/architecture**: used by Swing components and also a popular pattern.
    8. **Command pattern**: used by Swing action architecture and also a popular design pattern.
    9. **Strategy design pattern**: A popular design pattern used by AWT layout managers.

Refer **Q11** in "**How would you go about**…" section for a detailed discussion and code samples on GOF (Gang of Four) design patterns.

Recommended reading on design patterns:

- The famous Gang of Four book: Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addiso-Wesley Publishing Co., 1995; ISBN: 0201633612).

Tips:

- **Try to find out the needs of the project in which you will be working and the needs of the people within the project.**
- **80% of the interview questions are based on your own resume.**
- **Where possible briefly demonstrate how you applied your skills/knowledge in the key areas [design concepts, transactional issues, performance issues, memory leaks etc] as described in this book. Find the right time to raise questions and answer those questions to show your strength.**
- **Be honest to answer technical questions, you are not expected to remember everything (for example you might know a few design patterns but not all of them etc). If you have not used a design pattern in question, request the interviewer, if you could describe a different design pattern.**
- **Do not be critical, focus on what you can do. Also try to be humorous to show your smartness.**
- **Do not act superior.**

SECTION TWO

Enterprise Java – Interview questions & answers

KEY AREAS

- Specification Fundamentals SF
- Design Concepts DC
- Design Patterns DP
- Concurrency Issues CI
- Performance Issues PI
- Memory Issues MI
- Exception Handling EH
- Transactional Issues TI
- Security SE
- Scalability Issues SI
- Best Practices BP
- Coding[1] CO

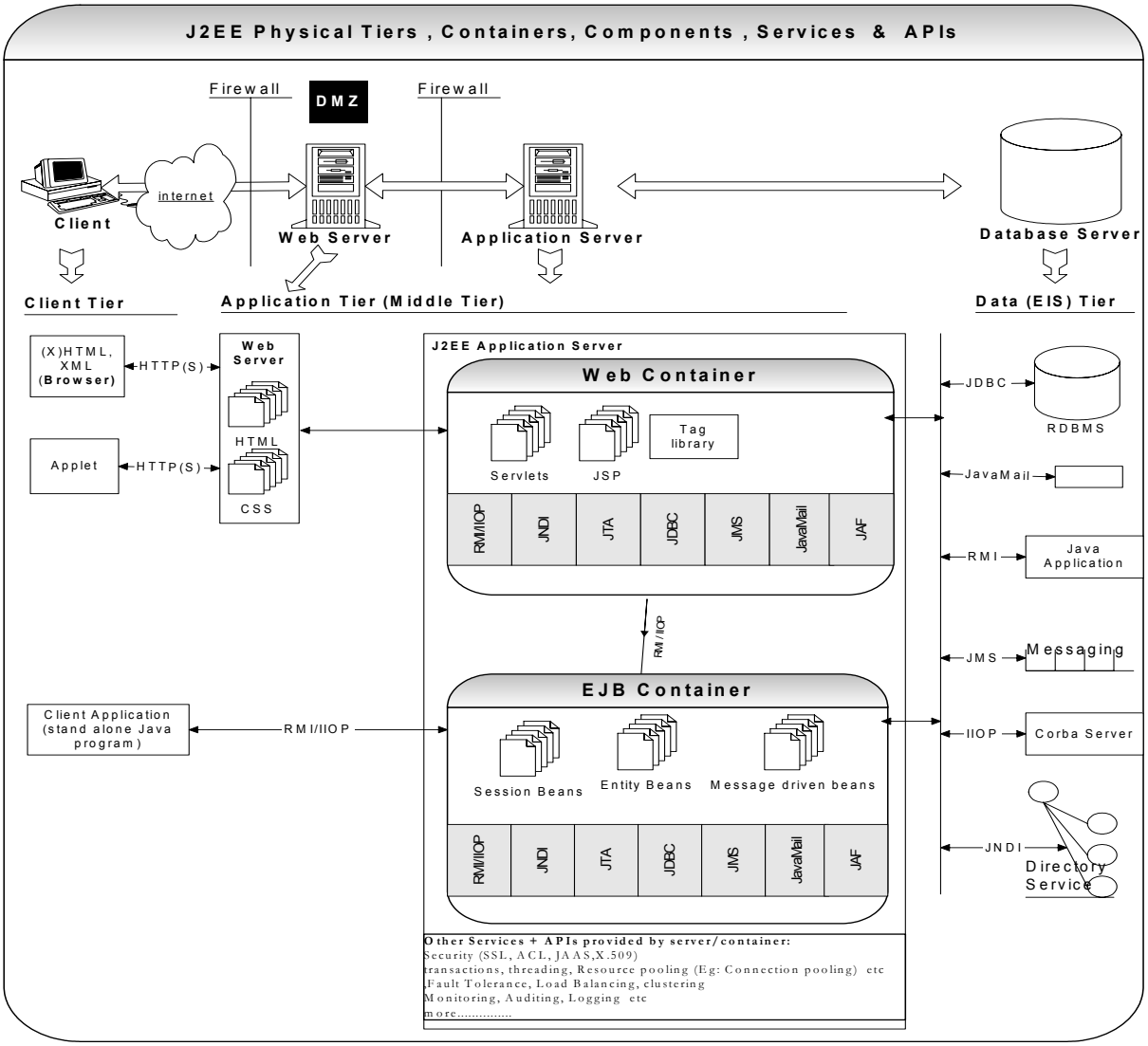
Popular Questions: Q02, Q03, Q10, Q16, Q19, Q20, Q24, Q25, Q30, Q31, Q36, Q39, Q40, Q45, Q46, Q48, Q49, Q53, Q58, Q63, Q64, Q65, Q66, Q71, Q72, Q73, Q76, Q77, Q78, Q79, Q83, Q84, Q85, Q86, Q87, Q89, Q90, Q91, Q93, Q96, Q97, Q98, Q100, Q102, Q106, Q107, Q110, Q123, Q124, Q125, Q129, **Q131**, Q136.

[1] Unlike other key areas, the CO is not always shown against the question but shown above the actual subsection of relevance within a question.

Enterprise - J2EE

Q 01: What is J2EE? What are J2EE components and services? SF

A 01: J2EE (**Java 2 E**nterprise **E**dition) is an environment for developing and deploying enterprise applications. The J2EE platform consists of J2EE components, services, Application Programming Interfaces (APIs) and protocols that provide the functionality for developing multi-tiered and distributed Web based applications.



A **J2EE component** is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and communicates with other components. The J2EE specification defines the following J2EE components:

Component type	Components	Packaged as
Applet	applets	JAR (**J**ava **AR**chive)
Application client	Client side Java codes.	JAR (**J**ava **AR**chive)
Web component	JSP, Servlet	WAR (**W**eb **AR**chive)
Enterprise JavaBeans	Session beans, Entity beans, Message driven beans	JAR (EJB Archive)
Enterprise application	WAR, JAR, etc	EAR (**E**nterprise **AR**chive)
Resource adapters	Resource adapters	RAR (**R**esource Adapter **AR**chive)

So what is the difference between a component and a service you may ask? A component is an application level software unit as shown in the table above. All the J2EE components depend on the container for the system level support like transactions, security, pooling, life cycle management, threading etc. A service is a component

that can be used remotely through a remote interface either synchronously or asynchronously (e.g. Web service, messaging system, sockets, RPC etc).

Containers (Web & EJB containers) are the interface between a J2EE component and the low level platform specific functionality that supports J2EE components. Before a Web, enterprise bean (EJB), or application client component can be executed, it must be assembled into a J2EE module (jar, war, and/or ear) and deployed into its container.

A J2EE server provides **system level support services** such us security, transaction management, JNDI (Java Naming and Directory Interface) lookups, remote access etc. J2EE architecture provides configurable and non-configurable services. The configurable service enables the J2EE components within the same J2EE application to behave differently based on where they are deployed. For example the security settings can be different for the same J2EE application in two different production environments. The non-configurable services include enterprise bean (EJB) and servlet life cycle management, resource pooling etc.

Protocols are used for access to Internet services. J2EE platform supports HTTP (HyperText Transfer Protocol), TCP/IP (Transmission Control Protocol / Internet Protocol), RMI (Remote Method Invocation), SOAP (Simple Object Access Protocol) and SSL (Secured Socket Layer) protocol.

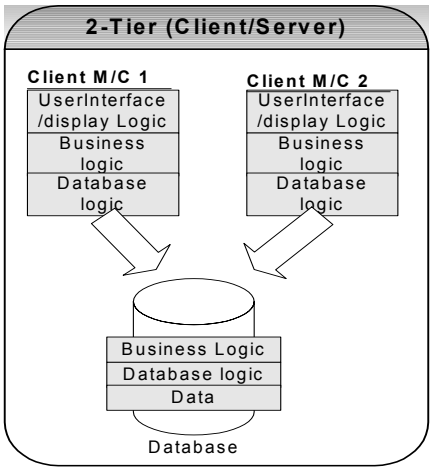
The J2EE API can be summarised as follows:

J2EE technology category	API (Application Program Interface)
Component model technology	**Java Servlet**, JavaServer Pages(**JSP**), Enterprise JavaBeans(**EJB**).
Web services technology	**JAXP** (Java API for XML Processing), **JAXR** (Java API for XML Registries), **SAAJ** (SOAP with attachment API for Java), **JAX-RPC** (Java API for XML-based RPC), **JAX-WS** (Java API for XML-based Web Services).
Other	**JDBC** (Java Database Connectivity), **JNDI** (Java Naming and Directory Interface), **JMS** (Java Messaging Service), **JCA** (J2EE Connector Architecture), **JTA** (Java Transaction API), **JavaMail**, **JAF** (JavaBeans Activation Framework – used by JavaMail), **JAAS** (Java Authentication and Authorization Service), **JMX** (Java Management eXtenstions).

Q 02: Explain the J2EE 3-tier or n-tier architecture? SF DC

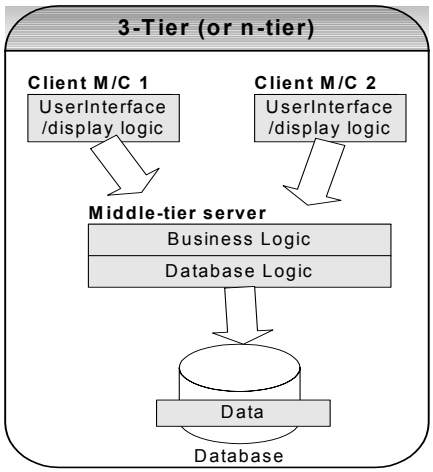
A 02: This is a very commonly asked question. Be prepared to draw some diagrams on the board. The J2EE platform is a multi-tiered system. A tier is a logical or functional partitioning of a system.

2 – tier system



When the developers are not disciplined, The display logic, business logic and database logic are muddled up and/or duplicated in a 2-tier client server system.

3 – tier system



The advantages of the multi-tier architecture are:

- Forced separation of user interface logic and business logic.
- Business logic sits on small number of centralized machines (may be just one).
- Easy to maintain, to manage, to scale, loosely coupled etc.

Each tier is assigned a unique responsibility in a 3-tier system. Each tier is logically separated and loosely coupled from each other, and may be distributed.

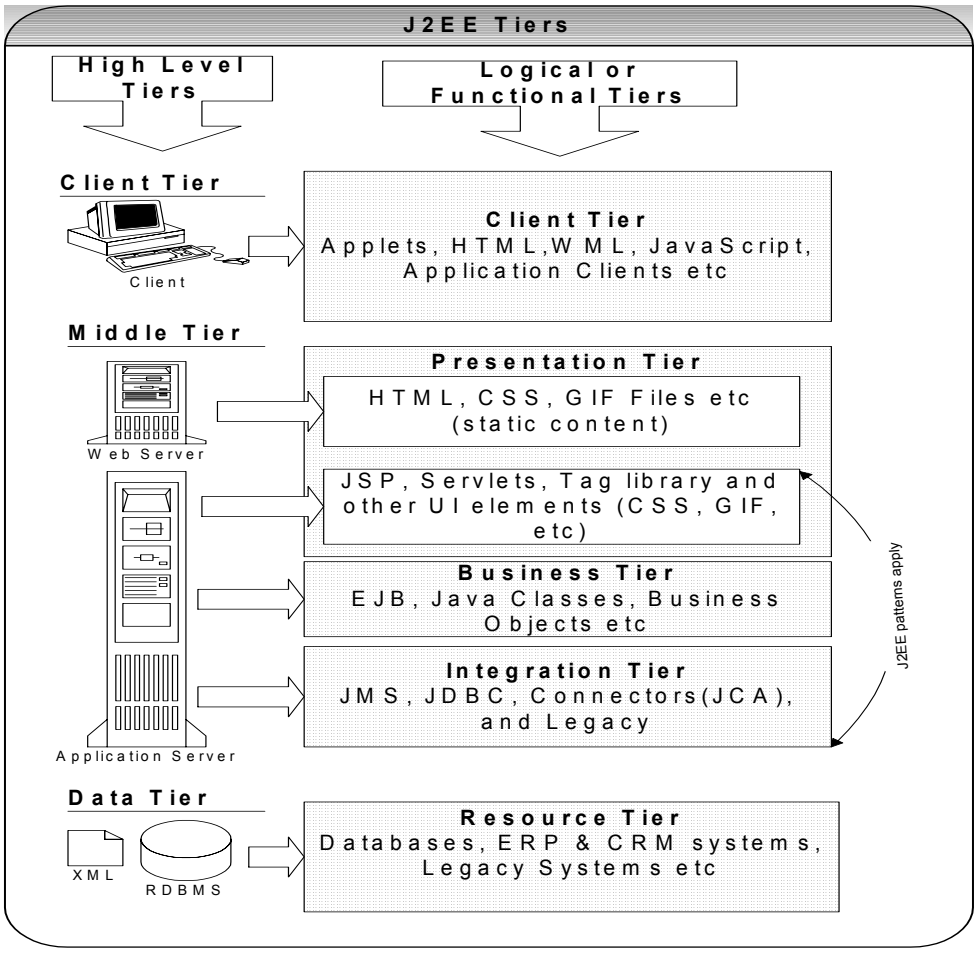
Client tier represents Web browser, a Java or other application, Applet, WAP phone etc. The client tier makes requests to the Web server who will be serving the request by either returning static content if it is present in the Web server or forwards the request to either Servlet or JSP in the application server for either static or dynamic content.

Presentation tier encapsulates the presentation logic required to serve clients. A Servlet or JSP in the presentation tier intercepts client requests, manages logons, sessions, accesses the business services, and finally constructs a response, which gets delivered to client.

Business tier provides the business services. This tier contains the business logic and the business data. All the business logic is centralised into this tier as opposed to 2-tier systems where the business logic is scattered between the front end and the backend. The benefit of having a centralised business tier is that same business logic can support different types of clients like browser, WAP, other stand-alone applications etc.

Integration tier is responsible for communicating with external resources such as databases, legacy systems, ERP systems, messaging systems like MQSeries etc. The components in this tier use JDBC, JMS, J2EE Connector Architecture (JCA) and some proprietary middleware to access the resource tier.

Resource tier is the external resource such as a database, ERP system, Mainframe system etc responsible for storing the data. This tier is also known as Data Tier or EIS (Enterprise Information System) Tier.



Note: On a high level J2EE can be construed as a **3-tier** system consisting of **Client Tier, Middle Tier** (or Application Tier) and **Data Tier**. But logically or functionally J2EE is a multi-tier (or n-tier) platform.

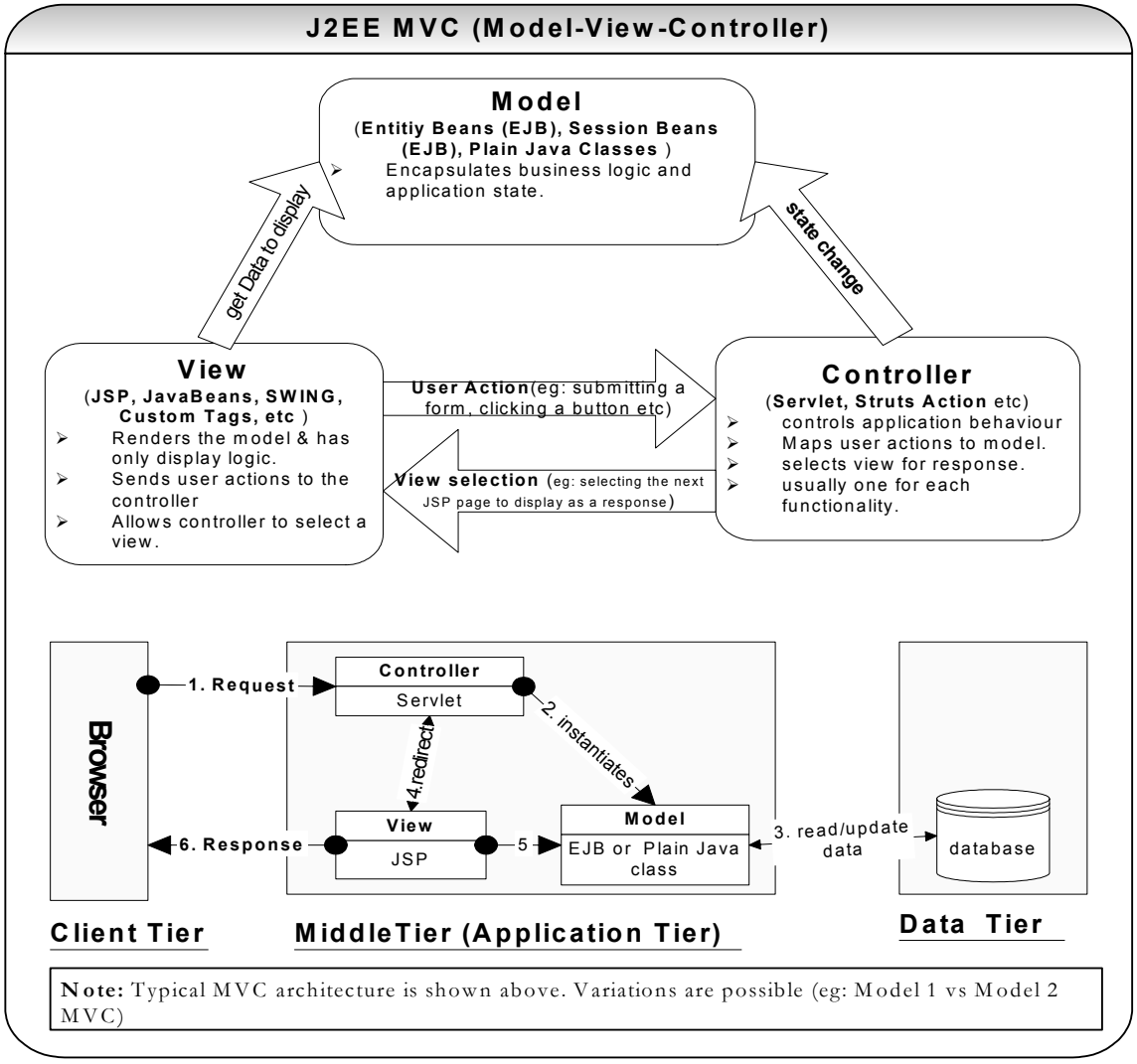
The advantages of a 3-tiered or n-tiered application: 3-tier or multi-tier architectures force separation among presentation logic, business logic and database logic. Let us look at some of the key benefits:

- **Manageability:** Each tier can be monitored, tuned and upgraded independently and different people can have clearly defined responsibilities.

- **Scalability:** More hardware can be added and allows clustering (i.e. horizontal scaling).
- **Maintainability:** Changes and upgrades can be performed without affecting other components.
- **Availability:** Clustering and load balancing can provide availability.
- **Extensibility:** Additional features can be easily added.

Q 03: Explain MVC architecture relating to J2EE? DC DP

A 03: This is also a very popular interview question. MVC stands for Model-View-Controller architecture. It divides the functionality of displaying and maintaining of the data to minimise the degree of coupling (i.e. promotes loose coupling) between components.



A **model** represents the **core business logic** and **state**. A model commonly maps to data in the database and will also contain core business logic.

A **View** renders the contents of a model. A view accesses the data from the model and adds **display logic** to present the data.

A **Controller** acts as the **glue between a model and a view**. A controller delegates the request to the model for application logic and state and also centralises the logic for dispatching the request to the next view based on the input parameters from the client and the application state. A controller also decouples JSP pages and the Servlet by handling the view selection.

Q 04: How to package a module, which is, shared by both the WEB and the EJB modules? **SF**

A 04: Package the modules shared by both WEB and EJB modules as dependency jar files. Define the **Class-Path:** property in the **MANIFEST.MF** file in the EJB jar and the Web war files to refer to the shared modules. [Refer **Q7** in Enterprise section for diagram: *J2EE deployment structure*].

The **MANIFEST.MF** files in the EJB jar and WEB war modules should look like:

```
Manifest-Version: 1.0
Created-By: Apache Ant 1.5
Class-Path: myAppsUtil.jar
```

Q 05: Why use design patterns in a J2EE application? **DP**

A 05:

- **They have been proven.** Patterns reflect the experience and knowledge of developers who have successfully used these patterns in their own work. It lets you leverage the collective experience of the development community.

  Example Session facade and value object patterns evolved from performance problems experienced due to multiple network calls to the EJB tier from the WEB tier. Fast lane reader and Data Access Object patterns exist for improving database access performance. The flyweight pattern improves application performance through object reuse (which minimises the overhead such as memory allocation, garbage collection etc).

- **They provide common vocabulary.** Patterns provide software designers with a common vocabulary. Ideas can be conveyed to developers using this common vocabulary and format.

  Example Should we use a Data Access Object (DAO)? How about using a Business Delegate? Should we use Value Objects to reduce network overhead? Etc.

Q 06: What is the difference between a Web server and an application server? **SF**

A 06:

Web Server	Application Server
Supports HTTP protocol. When the Web server receives an HTTP request, it responds with an HTTP response, such as sending back an HTML page (static content) or delegates the dynamic response generation to some other program such as CGI scripts or Servlets or JSPs in the application server.	Exposes **business logic** and **dynamic content** to the client through various protocols such as HTTP, TCP/IP, IIOP, JRMP etc.
Uses various scalability and fault-tolerance techniques.	Uses various scalability and fault-tolerance techniques. In addition provides resource pooling, component life cycle management, transaction management, messaging, security etc. Provides services for components like Web container for servlet components and EJB container for EJB components.

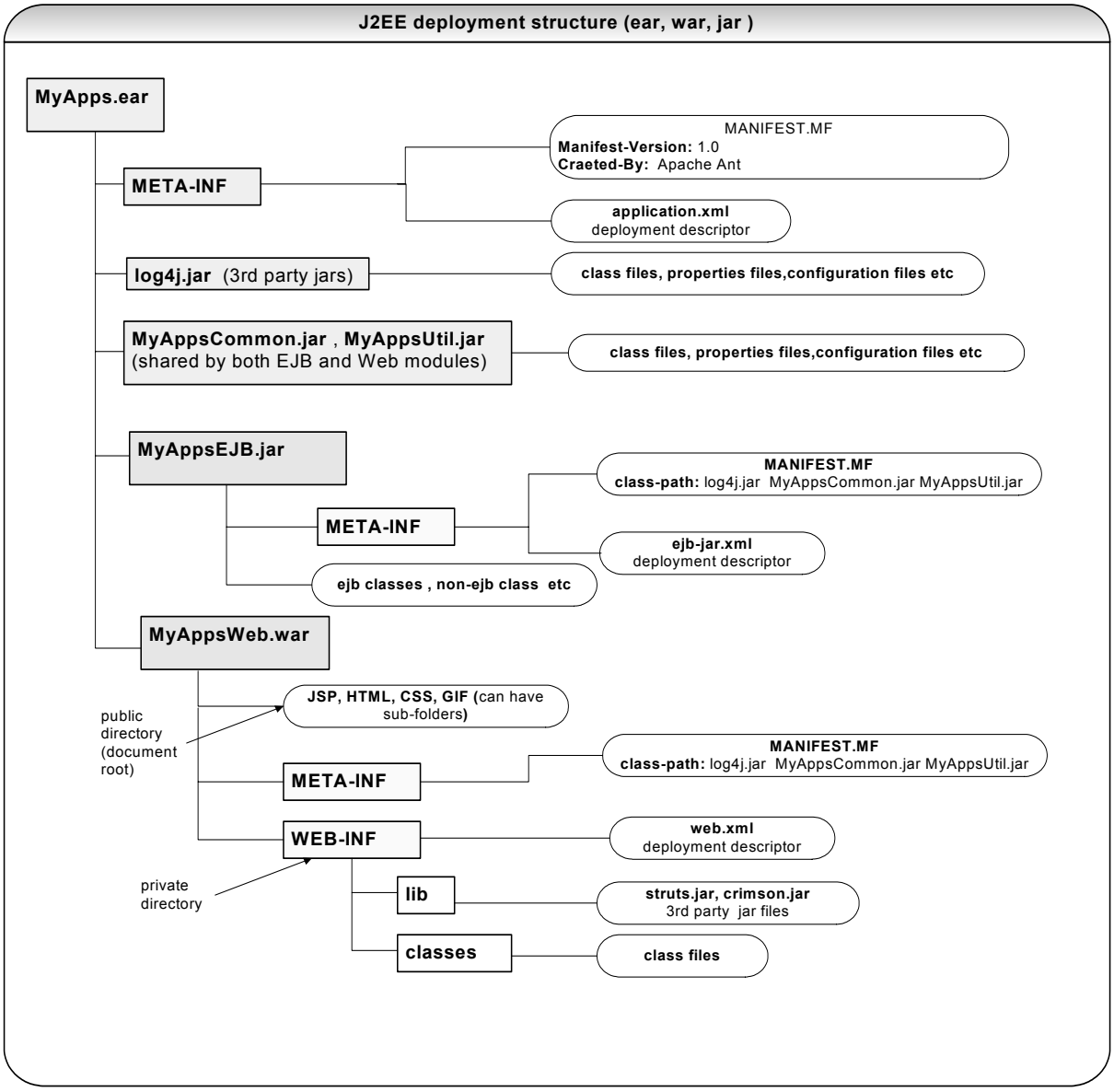
With the advent of XML Web services the line between application servers and Web servers is not clear-cut. By passing XML documents between request and response the Web server can behave like an application server.

Q 07: What are ear, war and jar files? What are J2EE Deployment Descriptors? **SF**

A 07: ear, war and jar are standard application deployment archive files. Since they are a standard, any application server (at least in theory) will know how to unpack and deploy them.

An EAR file is a standard JAR file with an ".ear" extension, named from Enterprise ARchive file. A J2EE application with all of its modules is delivered in EAR file. JAR files can't have other JAR files. But EAR and WAR (Web ARchive) files can have JAR files.

An EAR file contains all the JARs and WARs belonging to an application. JAR files contain the EJB classes and **WAR** files contain the Web components (JSPs, static content (HTML, CSS, GIF etc), Servlets etc.). The J2EE application client's class files are also stored in a JAR file. EARs, JARs, and WARs all contain an XML-based deployment descriptor.



Deployment Descriptors

A deployment descriptor is an XML based text file with a ".xml" extension that describes a component's deployment settings. A J2EE application and each of its modules has its own deployment descriptor. Pay attention to elements marked in bold in the sample deployment descriptor files shown below.

- **application.xml:** is a standard J2EE deployment descriptor, which includes the following structural information: EJB jar modules, WEB war modules, <security-role> etc. Also since EJB jar modules are packaged as jars the same way dependency libraries like log4j.jar, commonUtil.jar etc are packaged, the application.xml descriptor will distinguish between these two jar files by explicitly specifying the EJB jar modules.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN"
                        "http://java.sun.com/j2ee/dtds/application_1_2.dtd">
<application id="Application_ID">
   <display-name>MyApps</display-name>
   <module id="EjbModule_1">
      <ejb>MyAppsEJB.jar</ejb>
   </module>

   <module id="WebModule_1">
      <web>
```

```
            <web-uri>MyAppsWeb.war</web-uri>
            <context-root>myAppsWeb</context-root>
        </web>
    </module>

    <security-role id="SecurityRole_1">
        <description>Management position</description>
        <role-name>managger</role-name>
    </security-role>
</application>
```

- **ejb-jar.xml**: is a standard deployment descriptor for an EJB module.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
                         "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar id="ejb-jar_ID">
    <display-name>MyAppsEJB</display-name>

    <enterprise-beans>
        <session id="ContentService">
            <ejb-name>ContentService</ejb-name>
            <home>ejb.ContentServiceHome</home>
            <remote>ejb.ContentService</remote>
            <ejb-class>ejb.ContentServiceBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Bean</transaction-type>
        </session>

        <entity>
            <ejb-name>Bid</ejb-name>
            <home>ejb.BidHome</home>
            <remote>ejb.Bid</remote>
            <ejb-class>ejb.BidBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>ejb.BidPK</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>bid</field-name></cmp-field>
            <cmp-field><field-name>bidder</field-name></cmp-field>
            <cmp-field><field-name>bidDate</field-name></cmp-field>
            <cmp-field><field-name>id</field-name></cmp-field>
        </entity>
    </enterprise-beans>

    <!-- OPTIONAL -->

    <assembly-descriptor>

        <!-- OPTIONAL, can be many -->
        <security-role>
            <description>
                Employee is allowed to ...
            </description>
            <role-name>employee</role-name>
        </security-role>

        <!-- OPTIONAL.  Can be many -->
        <method-permission>
            <!-- Define role name in "security-role" -->
            <!-- Must be one or more -->
            <role-name>employee</role-name>
            <!-- Must be one or more -->
            <method>
                <ejb-name>ContentService</ejb-name>
                <!-- * = all methods -->
                <method-name>*</method-name>
            </method>

            <method>
                <ejb-name>Bid</ejb-name>
                <method-name>findByPrimaryKey</method-name>
            </method>
        </method-permission>
```

```
    <!-- OPTIONAL, can be many.  How the container is to manage
      transactions when calling an EJB's business methods -->

    <container-transaction>
      <!-- Can specify many methods at once here -->
      <method>
        <ejb-name>Bid</ejb-name>
        <method-name>*</method-name>
      </method>
      <!-- NotSupported|Supports|Required|RequiresNew|Mandatory|Never -->
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>

</ejb-jar>
```

- **web.xml**: is a standard deployment descriptor for a WEB module.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
                         "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
 <display-name>myWebApplication</display-name>
 <context-param>
   <param-name>GlobalContext.ClassName</param-name>
   <param-value>web.GlobalContext</param-value>
 </context-param>

 <servlet>
  <servlet-name>MyWebController</servlet-name>
       <servlet-class>web.MyWebController</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/config/myConfig.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
  <servlet-name>MyWebController</servlet-name>
  <url-pattern>/execute/*</url-pattern>
 </servlet-mapping>

 <error-page>
   <error-code>400</error-code>
   <location>/WEB-INF/jsp/errors/myError.jsp</location>
 </error-page>

 <taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/lib/taglib/struts/struts-bean.tld</taglib-location>
 </taglib>

 <security-constraint>
  <web-resource-collection>
    <web-resource-name>Employer</web-resource-name>
    <description></description>
    <url-pattern>/execute/employ</url-pattern>
    <http-method>POST</http-method>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description></description>
    <role-name>advisor</role-name>
  </auth-constraint>
 </security-constraint>

 <login-config>
  <auth-method>FORM</auth-method>
  <realm-name>FBA</realm-name>
  <form-login-config>
    <form-login-page>/execute/MyLogon</form-login-page>
```

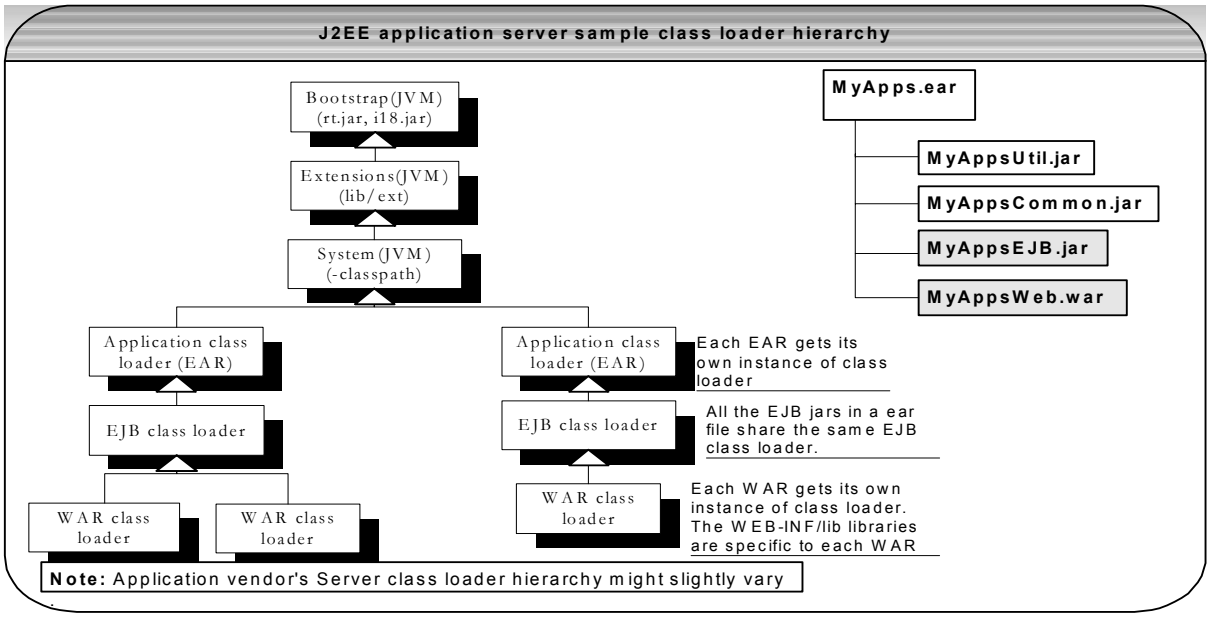
```
    <form-error-page>/execute/MyError</form-error-page>
  </form-login-config>
</login-config>

<security-role>
  <description>Advisor</description>
  <role-name>advisor</role-name>
</security-role>

</web-app>
```

Q 08: Explain J2EE class loaders? **SF**

A 08: J2EE application server sample class loader hierarchy is shown below. (Also refer to **Q4** in Java section). As per the diagram the J2EE application specific class loaders are children of the "*System –classpath"* class loader. When the parent class loader is above the "*System –Classpath"* class loader in the hierarchy as shown in the diagram (i.e. bootstrap class loader or extensions class loader) then child class loaders implicitly have visibility to the classes loaded by its parents. When a parent class loader is below a "*System -Classpath"* class loader then the child class loaders will only have visibility into the classes loaded by its parents **only if they are explicitly specified in a manifest file** (MANIFEST.MF) of the child class loader.

Example As per the diagram, if the EJB module *MyAppsEJB.jar* wants to refer to *MyAppsCommon.jar* and *MyAppsUtil.jar* we need to add the following entry in the MyAppsEJB.jar's manifest file MANIFEST.MF.

class-path: MyAppsCommon.jar MyAppsUtil.jar



This is because the application (EAR) class loader loads the *MyAppsCommon.jar* and *MyAppsUtil*.jar. The EJB class loader loads the *MyAppsEJB.jar*, which is the child class loader of the application class loader. The WAR class loader loads the MyAppsWeb.war.

Every J2EE application or EAR gets its own instance of the application class loader. This class loader is responsible for loading all the dependency jar files, which are shared by both WEB and EJB modules. For example third party libraries like log4j, utility classes, shared classes or common classes (Exception thrown by an EJB module should be caught by a WEB module) etc.

The key difference between the EJB and WAR class loader is that all the EJB jars in the application **share the same EJB class loader** whereas WAR files get their own class loader. This is because the EJBs have inherent relationship between one another (ie EJB-EJB communication between EJBs in different applications but hosted on the same JVM) but the Web modules do not. Every WAR file should be able to have its own WEB-INF/lib third party libraries and need to be able to load its own version of converted logon.jsp Servlet so each WEB module is isolated in its own class loader.

So if two different WEB modules want to use two different versions of the same EJB then we need to have two different ear files. As was discussed in the **Q4** in Java section the class loaders use a **delegation model** where the child class loaders delegate the loading up the hierarchy to their parent before trying to load it itself only if the parent can't load it. But with regards to WAR class loaders, some application servers provide a setting to turn this behaviour off (DelegationMode=false). This delegation mode is recommended in the Servlet 2.3 specification.

As a general rule **classes should not be deployed higher in the hierarchy than they are supposed to exist**. This is because if you move one class up the hierarchy then you will have to move other classes up the hierarchy as well. This is because classes loaded by the parent class loader can't see the classes loaded by its child class loaders (**uni-directional bottom-up visibility**).

Enterprise - Servlet

Q 09: What is the difference between CGI and Servlet? SF

Q 09:

Traditional CGI (Common Gateway Interface)	Java Servlet
Traditional CGI creates a heavy weight process to handle each http request. N number of copies of the same traditional CGI programs is copied into memory to serve N number of requests.	Spawns a lightweight Java thread to handle each http request. Single copy of a type of servlet but N number of threads (thread sizes can be configured in an application server).

In the Model 2 MVC architecture, servlets process requests and select JSP views. So servlets act as controller. Servlets intercept the incoming HTTP requests from the client (browser) and then dispatch the request to the business logic model (e.g. EJB, POJO - Plain Old Java Object, JavaBeans etc). Then select the next JSP view for display and deliver the view to client as the presentation (response). It is the best practice to use Web tier UI frameworks like Struts, JavaServer Faces etc, which uses proven and tested design patterns.

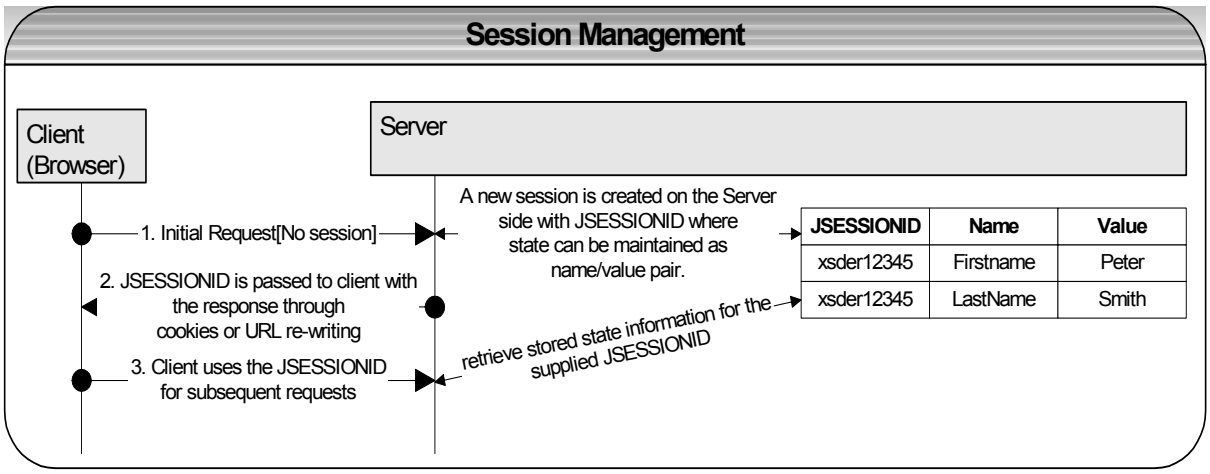
Q 10: HTTP is a stateless protocol, so how do you maintain state? How do you store user data between requests? SF PI BP

A 10: This is a commonly asked question as well. You can retain the state information between different page requests as follows:

HTTP Sessions are the recommended approach. A session identifies the requests that originate from the same browser during the period of conversation. All the servlets can share the same session. The JSESSIONID is generated by the server and can be passed to client through cookies, URL re-writing (if cookies are turned off) or built-in SSL mechanism. Care should be taken to **minimize size of objects stored in session** and **objects stored in session should be serializable.** In a Java servlet the session can be obtained as follows: CO

```
HttpSession session = request.getSession(); //returns current session or a new session
```

Sessions can be timed out (configured in web.xml) or manually invalidated.



Hidden Fields on the pages can maintain state and they are not visible on the browser. The server treats both hidden and non-hidden fields the same way.

```
<INPUT type="hidden" name="Firstname" value="Peter">
<INPUT type="hidden" name="Lastname" value="Smith">
```

The disadvantage of hidden fields is that they may expose sensitive or private information to others.

URL re-writing will append the state information as a query string to the URL**.** This should not be used to maintain private or sensitive information**.**

```
Http://MyServer:8080/MyServlet?Firstname=Peter&Lastname=Smith
```

Cookies: A cookie is a piece of text that a Web server can store on a user's hard disk. Cookies allow a website to store information on a user's machine and later retrieve it. These pieces of information are stored as name-value pairs. The cookie data moves in the following manner:

- If you type the URL of a website into your browser, your browser sends the request to the Web server. When the browser does this it looks on your machine for a cookie file that URL has set. If it finds it, your browser will send all of the name-value pairs along with the URL. If it does not find a cookie file, it sends no cookie data.

- The URL's Web server receives the cookie data and requests for a page. If name-value pairs are received, the server can use them. If no name-value pairs are received, the server can create a new ID and then sends name-value pairs to your machine in the header for the Web page it sends. Your machine stores the name value pairs on your hard disk.

Cookies can be used to determine how many visitors visit your site. It can also determine how many are new versus repeated visitors. The way it does this is by using a database. The first time a visitor arrives, the site creates a new ID in the database and sends the ID as a cookie. The next time the same user comes back, the site can increment a counter associated with that ID in the database and know how many times that visitor returns. The sites can also store user preferences so that site can look different for each visitor.

Which mechanism to choose?

Session mechanism	Description
HttpSession	▪ There is no limit on the size of the session data kept. ▪ The performance is good. ▪ This is the preferred way of maintaining state. If we use the HTTP session with the application server's persistence mechanism (server converts the session object into BLOB type and stores it in the Database) then the performance will be moderate to poor. **Note**: When using HttpSession mechanism you need to take care of the following points: ▪ Remove session explicitly when you no longer require it. ▪ Set the session timeout value. ▪ Your application server may serialize session objects after crossing a certain memory limit. This is expensive and affects performance. So decide carefully what you want to store in a session.
Hidden fields	▪ There is no limit on size of the session data. ▪ May expose sensitive or private information to others (So not good for sensitive information). ▪ The performance is moderate.
URL rewriting	▪ There is a limit on the size of the session data. ▪ Should not be used for sensitive or private information. ▪ The performance is moderate.
Cookies	▪ There is a limit for cookie size. ▪ The browser may turn off cookies. ▪ The performance is moderate. The benefit of the cookies is that state information can be stored regardless of which server the client talks to and even if all servers go down. Also, if required, state information can be retained across sessions.

Q 11: Explain the life cycle methods of a servlet? **SF**

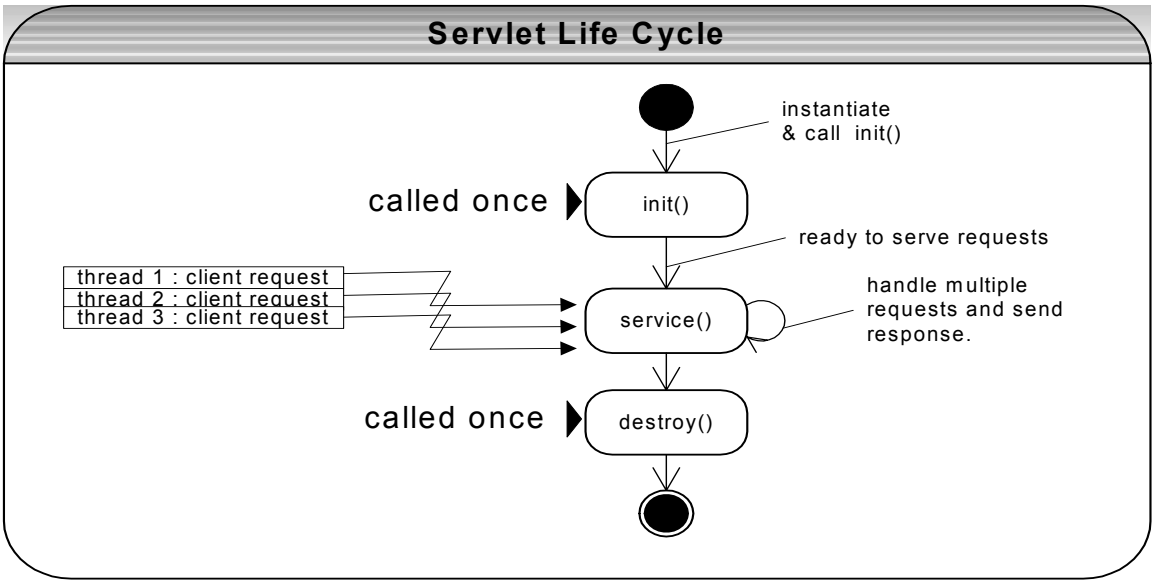
A 11: The Web container is responsible for managing the servlet's life cycle. The Web container creates an instance of the servlet and then the container calls the init() method. At the completion of the init() method the servlet is in

ready state to service requests from clients. The container calls the servlet's service() method for handling each request by spawning a new thread for each request from the Web container's thread pool [It is also possible to have a single threaded Servlet, refer **Q16** in Enterprise section]. Before destroying the instance the container will call the destroy() method. After destroy() the servlet becomes the potential candidate for garbage collection.

Note on servlet reloading:

Most servers can reload a servlet after its class file has been modified provided the servlets are deployed to **$server_root/servlets** directory. This is achieved with the help of a custom class loader. This feature is handy for development and test phases. This is not recommended for production since it can degrade performance because of timestamp comparison for each request to determine if a class file has changed. So for production it is recommended to move the servlet to server's class path ie **$server_root/classes**.

When a server dispatches a request to a servlet, the server first checks if the servlet's class file has changed on disk. If it has changed, the server abandons the class loader used to load the old version and creates a new instance of the custom class loader to load the new version. Old servlet versions can stay in memory indefinitely (so the effect is the other classes can still hold references to the old servlet instances, causing odd side effects, but the old versions are not used to handle any more requests. Servlet reloading is not performed for classes found in the server's classpath because the core, primordial class loader, loads those classes. These classes are loaded once and retained in memory even when their class files change.



Q 12: Explain the directory structure of a WEB application? SF SE

A 12: Refer **Q7** in Enterprise section for diagram: *J2EE deployment structure* and explanation in this section where *MyAppsWeb.war* is depicting the Web application directory structure. The directory structure of a Web application consists of two parts:

- A **public** resource directory (**document root**): The document root is where JSP pages, client-side classes and archives, and static Web resources are stored.
- A **private** directory called WEB-INF: which contains following files and directories:
  - **web.xml** : Web application deployment descriptor.
  - ***.tld** : Tag library descriptor files.
  - **classes** : A directory that contains server side classes like servlets, utility classes, JavaBeans etc.
  - **lib** : A directory where JAR (archive files of tag libraries, utility libraries used by the server side classes) files are stored.

Note: JSP resources usually reside directly or under subdirectories of the **document root,** which **are directly accessible** to the user through the URL. If you want to protect your Web resources then hiding the JSP files behind the WEB-INF directory can protect the JSP files from direct access. Refer **Q35** in Enterprise section.

Q 13: What is the difference between doGet () and doPost () or GET and POST? SF SE

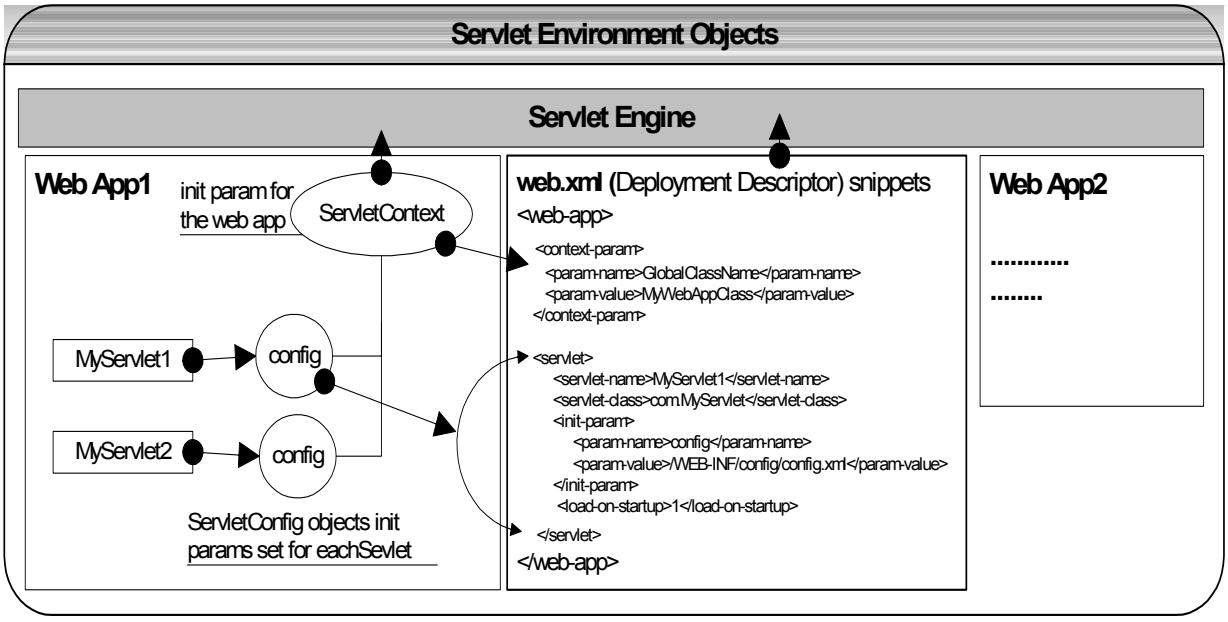
A 13:

GET or doGet()	POST or doPost()
The request parameters are transmitted as a query string appended to the request. Allows browser bookmarks but not appropriate for transmitting private or sensitive information. http://MyServer/MyServlet?name=paul This is a security risk.	The request parameters are passed with the body of the request. More secured.
GET was originally intended for static resource retrieval.	POST was intended for input data submits where the results are expected to change.
GET is not appropriate when large amounts of input data are being transferred.	

Q 14: What are the ServletContext and ServletConfig objects? What are Servlet environment objects? SF

A 14: The Servlet Engine uses both objects.

ServletConfig	ServletContext
The ServletConfig parameters are for **a particular Servlet**. The parameters are specified in the web.xml (ie deployment descriptor).	The ServletContext parameters are specified for the **entire Web application**. The parameters are specified in the web.xml (ie deployment descriptor).



Q 15: What is the difference between HttpServlet and GenericServlet? SF

A 15:

GenericServlet	HttpServlet
A GenericServlet has a service() method to handle requests.	The HttpServlet extends GenericServlet and adds support for HTTP protocol based methods like doGet(), doPost(), doHead() etc.
Protocol independent.	Protocol dependent.

Q 16: How do you make a Servlet thread safe? What do you need to be concerned about with storing data in Servlet instance fields? CI PI BP

A 16: As shown in the figure *Servlet Life Cycle* in **Q11** in Enterprise section, a typical (or default) Servlet life cycle creates a single instance of each servlet and creates multiple threads to handle the service() method. **The multi-threading aids efficiency but the servlet code must be coded in a thread safe manner.** The shared resources (e.g. instance variables, utility or helper objects etc) should be appropriately synchronized or should only use variables in a read-only manner. Having large chunks of code in synchronized blocks in your service methods can adversely affect performance and makes the code more complex.

Alternatively it is possible to have a **single threaded model of a servlet** by implementing the marker or null interface javax.servlet.SingleThreadedModel. The container will use one of the following approaches to ensure thread safety:

- **Instance pooling** where container maintains a pool of servlets.
- **Sequential processing** where new requests will wait while the current request is being processed.

Best practice: It is best practice to use multi-threading and stay away from the **single threaded model of the servlet** unless otherwise there is a compelling reason for it. Shared resources can be synchronized or used in read-only manner or shared values can be stored in a database table. The single threaded model can adversely affect performance.

Q 17: What is pre-initialization of a Servlet? LF

A 17: By default the container does not initialize the servlets as soon as it starts up. It initializes a servlet when it receives a request for the first time for that servlet. This is called **lazy loading**. The servlet deployment descriptor (web.xml) defines the <load-on-startup> element, which can be configured to make the servlet container load and initialize the servlet as soon as it starts up. The process of loading a servlet before any request comes in is called **pre-loading** or **pre-initializing** a servlet. We can also specify the order in which the servlets are initialized.

```
<load-on-startup>2</load-on-startup>
```

Q 18: What is a RequestDispatcher? What object do you use to forward a request? LF CO

A 18: A Servlet can obtain its **RequestDispatcher** object from its *ServletContext*.

```
//…inside the doGet() method
ServletContext sc = getServletContext();
RequestDispatcher rd = sc.getRequestDispatcher(url);

// forwards the control to another servlet or JSP to generate response.  This method allows one servlet to do preliminary
//processing of a request and another resource to generate the response
rd.forward(request,response);
   or
// includes the content of the resource such as Servlet, JSP, HTML, Images etc into the calling Servlet's response.
rd.include(request, response);
```

Q 19: What is the difference between forwarding a request and redirecting a request? LF DC

A 19: Both methods redirect you to a new resource like Servlet, JSP etc. But

redirecting - sendRedirect()	forward
Sends a header back to the browser, which contains the name of the resource to be redirected to. The browser will make a **fresh request from this header information**. Need to provide absolute URL path.	Forward action takes place within the server **without the knowledge of the browser**.
Has an overhead of extra remote trip but has the advantage of being able to refer to any resource on the same or different domain and also allows book marking of the page.	No extra network trip.

Q 20: What are the considerations for servlet clustering? DC SI

A 20: The clustering promotes high availability and scalability. The considerations for servlet clustering are:

- **Objects stored in a session should be serializable** to support in-memory replication of sessions. Also consider the overhead of serializing very large objects. Test the performance to make sure it is acceptable.
- **Design for idempotence**. Failure of a request or impatient users clicking again can result in duplicate requests being submitted. So the Servlets should be able to tolerate duplicate requests.
- **Avoid using instance and static variables in read and write mode** because different instances may exist on different JVMs. Any state should be held in an external resource such as a database.
- **Avoid storing values in a ServletContext**. A ServletContext is not serializable and also the different instances may exist in different JVMs.
- **Avoid using java.io.* because the files may not exist on all backend machines**. Instead use getResourceAsStream().

Q 21: If an object is stored in a session and subsequently you change the state of the object, will this state change replicated to all the other distributed sessions in the cluster? **DC** **SI**

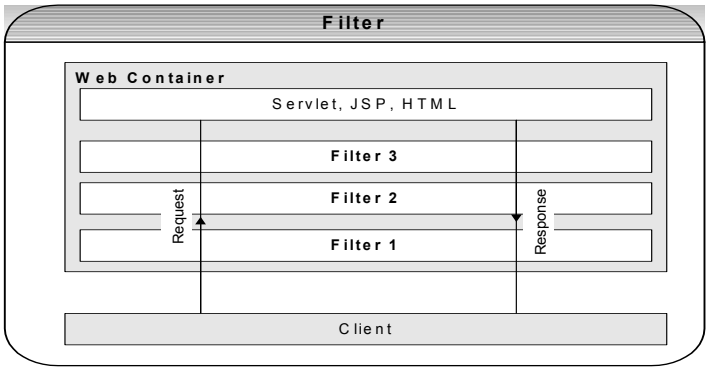
A 21: **No**. Session replication is the term that is used when your current service state is being replicated across multiple application instances. Session replication occurs when we replicate the information (ie **session attributes**) that are stored in your HttpSession. The container propagates the changes only when you call the ***setAttribute(……)*** method. So mutating the objects in a session and then by-passing the setAttribute(……..) will not replicate the state change. **CO**

Example If you have an ArrayList in the session representing shopping cart objects and if you just call **getAttribute()** to retrieve the ArrayList and then add or change something without calling the **setAttribute(……)** then the container may not know that you have added or changed something in the ArrayList. So the session will not be replicated.

Q 22: What is a filter, and how does it work? **LF** **DP**

A 22: A filter dynamically intercepts requests and responses to transform or use the information contained in the requests or responses but typically do not themselves create responses. Filters can also be used to transform the response from the Servlet or JSP before sending it back to client. Filters improve reusability by placing recurring tasks in the filter as a reusable unit.

A good way to think of Servlet filters is as a chain of steps that a request and response must go through before reaching a Servlet, JSP, or static resource such as an HTML page in a Web application.



The filters can be used for caching and compressing content, logging and auditing, image conversions (scaling up or down etc), authenticating incoming requests, XSL transformation of XML content, localization of the request and the response, site hit count etc. The filters are configured through the web.xml file as follows:

```
<web-app>
  <filter>
    <filter-name>HitCounterFilter</filter-name>
    <filter-class>myPkg.HitCounterFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>HitCounterFilter</filter-name>
    <url-pattern>/usersection/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

The *HitCounterFilter* will intercept the requests from the URL pattern */usersection* followed by any resource name.

Design Pattern: Servlet filters use the slightly modified version of the **chain of responsibility** design pattern. Unlike the classic (only one object in the chain handle the request) chain of responsibility where filters allow multiple objects (filters) in a chain to handle the request. If you want to modify the request or the response in the chain you can use the **decorator pattern** (Refer **Q11** in How would you go about… section).

Q 23: Explain declarative security for WEB applications? **SE**

A 23: Servlet containers implement declarative security. The administration is done through the deployment descriptor web.xml file. With **declarative security** the Servlets and JSP pages will be free from any security aware code. You can protect your URLs through web.xml as shown below:

```
web-app>
 <security-constraint>
   <web-resource-collection>
     <web-resource-name>PrivateAndSensitive</web-resource-name>
     <url-pattern>/private/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
     <role-name>executive</role-name>
     <role-name>admin</role-name>
   </auth-constraint>
 </security-constraint>

 <!-- form based authorization -->
 <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/login.jsp</form-login-page>
        <form-error-page>/error.jsp</form-error-page>
    </form-login-config>
 </login-config>
</web-app>
```

The user will be prompted for the configured login.jsp when restricted resources are accessed. The container also keeps track of which users have been previously authenticated.

Benefits: Very little coding is required and developers can concentrate on the application they are building and system administrators can administer the security settings without or with minimal developer intervention. Let's look at a sample programmatic security in a Web module like a servlet: **CO**

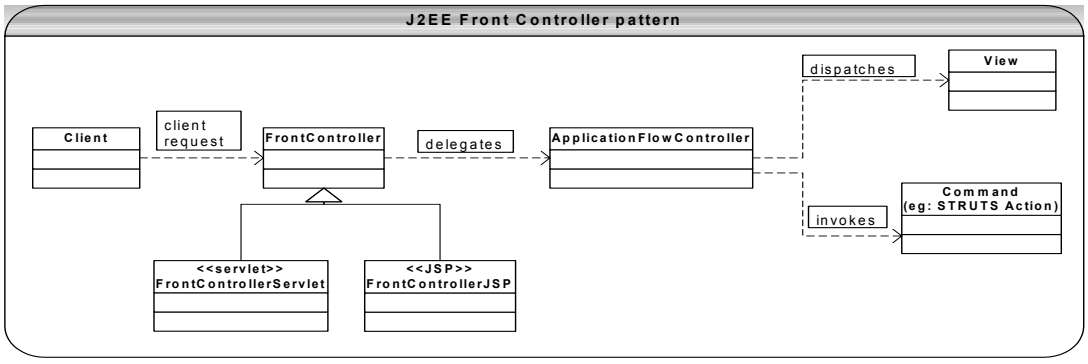
```
User user = new User();
Principal principal = request.getUserPrincipal();
if (request.isUserInRole("boss"))
   user.setRole(user.BOSS_ROLE);
```

Q 24: Explain the **Front Controller** design pattern or explain J2EE design patterns? **DP**

A 24: **Problem**: A J2EE system requires a centralized access point for HTTP request handling to support the integration of system services like security, data validation etc, content retrieval, view management, and dispatching. When the user accesses the view directly without going through a centralized mechanism, two problems may occur:

- Each view is required to provide its own system services often resulting in **duplicate code**.
- View navigation is left to the views. This may result in shared code for view content and view navigation.
- Distributed control is **more difficult to maintain**, since changes will often need to be made in numerous places.

Solution: Generally you write specific servlets for specific request handling. These servlets are responsible for data validation, error handling, invoking business services and finally forwarding the request to a specific JSP view to display the results to the user.



The **Front Controller** suggests that we **only have one Servlet** (instead of having specific Servlet for each specific request) centralising the handling of all the requests and delegating the functions like validation, invoking business services etc to a command or a helper component. For example Struts framework uses the command design pattern to delegate the business services to an action class.

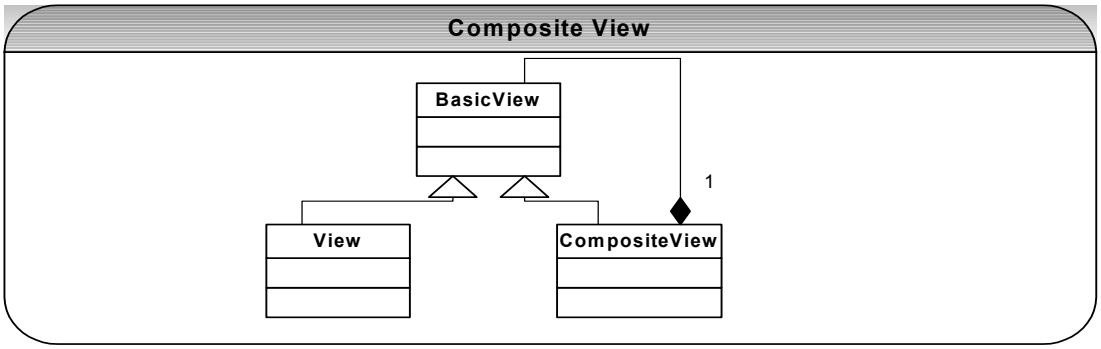
Benefits

- Avoid duplicating the control logic like security check, flow control etc.
- Apply the common logic, which is shared by multiple requests in the Front controller.
- Separate the system processing logic from the view processing logic.
- Provides a controlled and centralized access point for your system.

Q 25: Briefly discuss the following patterns Composite view, View helper, Dispatcher view and Service to worker? Or explain J2EE design patterns? **DP**

A 25:

- **Composite View:** Creates an aggregate view from atomic sub-views. The Composite View entirely focuses on the View. The View is typically a JSP page, which has the HTML, JSP Tags etc. The JSP display pages mostly have a side bar, header, footer and main content area. These are the sub-views of the view. The sub-views can be either static or dynamic. The best practice is to have these sub-views as separate JSP pages and include them in the whole view. This will enable **reuse of JSP sub-views and improves maintainability** by having to change them at one place only.



- **View Helper:** When processing logic is embedded inside the controller or view it causes code duplication in all the pages. This causes maintenance problems, as any change to piece of logic has to be done in all the views. In the view helper pattern the view delegates its processing responsibilities to its helper classes. The helper classes **JavaBeans:** used to compute and store the presentation data and **Custom Tags**: used for computation of logic and displaying them iteratively complement each other.

  Benefits Avoids embedding programming logic in the views and facilitates division of labour between Java developers and Web page designers.

- **Service to Worker** and **Dispatcher View:** These two patterns are a combination of Front Controller and View Helper patterns with a *dispatcher* component. One of the responsibilities of a Front Controller is choosing a view and dispatching the request to an appropriate view. This behaviour can be partitioned into a separate component known as a *dispatcher*. But these two patterns differ in the way they suggest different division of responsibility among the components.

Service to Worker	Dispatcher View
Combines the front controller (Refer **Q24** in Enterprise section) and dispatcher, with views and view helpers (refer **Q25** in Enterprise section) to handle client requests and dynamically prepares the response. ▪ Controllers delegate the content retrieval to the view helpers, which populates the intermediate model content for the view. ▪ Dispatcher is responsible for the view management and view navigation.	This pattern is structurally similar to the service to worker but the emphasis is on a different usage pattern. This combines the Front controller and the dispatcher with the view helpers but ▪ Controller **does not** delegate content retrieval to view helpers because this activity is deferred to view processing. ▪ Dispatcher is responsible for the view management and view navigation.

Promotes more up-front work by the front controller and dispatcher for the authentication, authorization, content retrieval, validation, view management and navigation.	Relatively has a **lightweight front controller** and dispatcher with minimum functionality **and most of the work is done by the view**.

Q 26: Explain Servlet URL mapping? SF

Q 26:

Servlet URL mapping

Without Mapping in web.xml

URL http://<hostname:port>/<webapp name>/servlet /<pathname>/<resourcename>

URL eg http://localhost:8080/myApps/**servlet**/myPath/MyServlet

File SERVER_HOME\WebApps\myApps**WEB-INF\Classes**\myPath\MyServlet

Server Root

Document root

With Mapping in web.xml deployment descriptor file

We can define the servlet mapping in the **web.xml** deploymnet descriptor file as shown below:

```
<web-app>
 <servlet>
   <servlet-name>MyServlet</servlet-name>
   <servlet-class>myPath.MyServlet</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>mine/*.do</url-pattern>
 </servlet-mapping>
<web-app>
```

URL after mapping http://localhost:8080/myApps/**mine/test.do**

Note: Which means every request which has a pattern of **http://localhost:8080/myApps/ mine/*.do** will be handled by the **myPath.MyServlet** class. (* denotes wild character for any alphanumeric name). Also possible to map MyServlet to the pattern of /mine/* , the * indicates any resource name followed by /mine.

How do we get the webapp name "myApps"

The webapp name is defined in the **application.xml** deployment descriptor file. The <context-root > denotes the web app nameas shown below

```
<application>
 ............
 <module id="WebModule_1">
   <web>
    <web-uri>myAppsWeb.war</web-uri>
    <context-root>myApps</context-root>
   </web>
 </module>
 ........
 <module id="EjbModule_1">
    <ejb>myEJB.jar</ejb>
 </module>
 .....
</application>
```

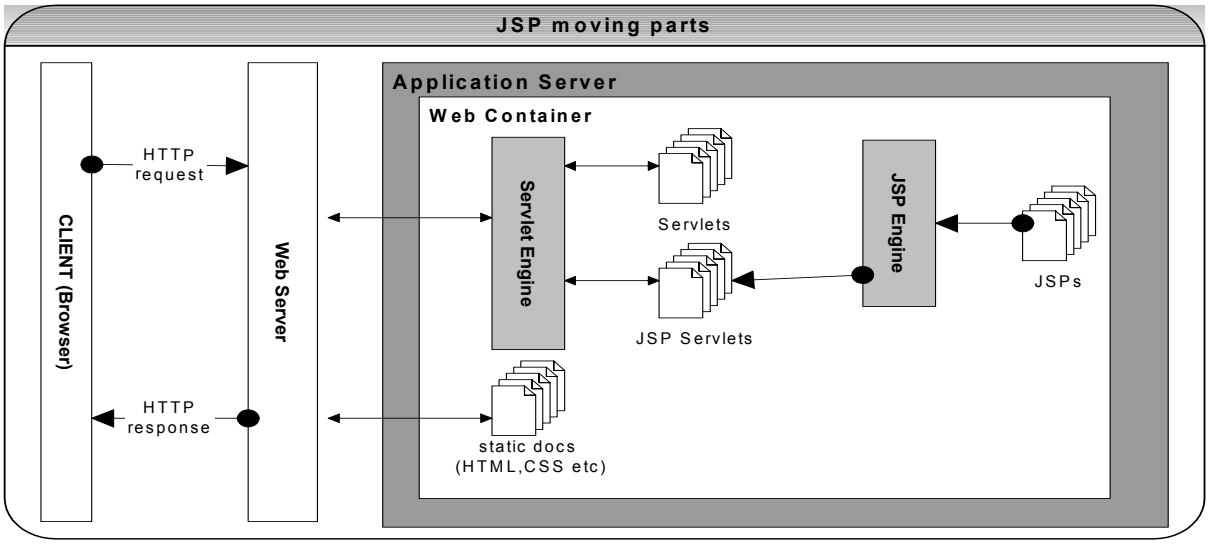
Enterprise - JSP

Q 27: What is a JSP? What is it used for? What do you understand by the term JSP translation phase or compilation phase? SF

A 27: JSP (Java ServerPages) is an extension of the Java Servlet technology. JSP is commonly used as the **presentation** layer for combining HTML and Java code. While Java Servlet technology is capable of generating HTML with out.println("<html>….. </html>") statements, where *out* is a *PrintWriter*. This **process of embedding HTML code with escape characters is cumbersome and hard to maintain**. The JSP technology solves this by providing a level of abstraction so that the developer can use custom tags and action elements, which can speed up Web development and are easier to maintain.

As shown in the figure the JSPs have a **translation** or a **compilation** process where the JSP engine translates and compiles a JSP file into a JSP Servlet. The translated and compiled JSP Servlet moves to the **execution phase (run time)** where they can handle requests and send response.

Unless explicitly compiled ahead of time, JSP files are compiled the first time they are accessed. On large production sites, or in situations involving complicated JSP files, compilation may cause unacceptable delays to users first accessing the JSP page. The JSPs can be compiled ahead of time (ie **precompiled**) using application server tools/settings or by writing your own script.



Q 28: Explain the life cycle methods of a JSP? SF

A 28:

- **Pre-translated:** Before the JSP file has been translated and compiled into the Servlet.
- **Translated:** The JSP file has been translated and compiled as a Servlet.
- **Initialized:** Prior to handling the requests in the service method the container calls the jspInit() to initialize the Servlet. Called only once per Servlet instance.
- **Servicing**: Services the client requests. Container calls this method for each request.
- **Out of service**: The Servlet instance is out of service. The container calls the jspDestroy() method.

Q 29: What are the main elements of JSP? What are scriplets? What are expressions? SF

A 29: There are two types of data in a JSP page.

- **Static part (**ie HTML, CSS etc**),** which gets copied directly to the response by the JSP Engine.
- **Dynamic part,** which contains anything that can be translated and compiled by the JSP Engine.

There are three types of dynamic elements. (**TIP:** remember **SAD** as an abbreviation for **S**cripting, **A**ction and **D**irective elements).

Scripting Elements: A JSP element that provides embedded Java statements. There are three types of scripting elements.

- **Declaration Element**: is the embedded Java declaration statement, which gets inserted at the Servlet class level.

```
<%! Calendar c = Calendar.getInstance(); %>
```

> ***Important***: declaring variables via this element is not thread-safe, because this variable ends up in the generated Servlet as an instance variable, not within the body of the _jspservice() method. Ensure their access is either read-only or synchronized.

- **Expression Element**: is the embedded Java expression, which gets evaluated by the service method.

```
<%= new Date()>
```

- **Scriptlet Elements**: are the embedded Java statements, which get executed as part of the service method. (**Note**: Not recommended to use Scriptlet elements because they don't provide reusability and maintainability. Use custom tags (like JSTL, JSF tags, etc) or beans instead).

```
<%
//Java codes
String userName=null;
userName=request.getParameter("userName");
%>
```

Action Elements: A JSP element that provides information for execution phase.

```
<jsp:useBean id="object_name" class="class_name"/>
<jsp:include page="scripts/login.jsp" />
```

Directive Elements: A JSP element that provides global information for the **translation** phase.

```
<%@ page import="java.util.Date" %>
<%@ include file="myJSP" %>
<%@ taglib uri="tagliburi" prefix="myTag"%>
```

Q 30: What are the different scope values or what are the different scope values for <jsp:usebean> ? SF

A 30:

Scope	Object	Comment
Page	PageContext	Available to the handling JSP page only.
Request	Request	Available to the handling JSP page or Servlet and forwarded JSP page or Servlet.
Session	Session	Available to any JSP Page or Servlet within the same session.
Application	Application	Available to all the JSP pages and Servlets within the same Web Application.

Q 31: What are the differences between static and a dynamic include? SF DC

A 31:

Static include <%@ include %>	Dynamic include <jsp:include …..>
During the translation or compilation phase all the included JSP pages are compiled into a single Servlet.	The dynamically included JSP is compiled into a separate Servlet. It is a separate resource, which gets to process the request, and the content generated by this resource is included in the JSP response.
No run time performance overhead.	Has run time performance overhead.

Which one to use: Use "static includes" when a JSP page does not change very often. For the pages, which change frequently, use dynamic includes. JVM has a 64kb limit on the size of the method and the entire JSP page is rendered as a single method**. If a JSP page is greater than 64kb, this probably indicates poor implementation**. When this method reaches its limit of 64kb it throws an error. This **error can be overcome by splitting the JSP files and including them dynamically** (i.e. using <jsp:include…….>) because the dynamic includes generate separate JSP Servlet for each included file.

Note: The "dynamic include" (jsp:include) has a **flush** attribute. This attribute indicates whether the buffer should be flushed before including the new content. In JSP 1.1 you will get an error if you omit this attribute. In JSP 1.2 you can omit this attribute because the flush attribute defaults to false.

Q 32: What are implicit objects and list them? SF

A 32: Implicit objects are the objects that are available for the use in JSP documents without being declared first. These objects are parsed by the JSP engine and inserted into the generated Servlet. The implicit objects are:

Implicit object	Scope	comment
request	Request	request
response	Page	response
pageContext	Page	page environment
session	Session	session
application	Application	same as ServletContext
out	Page	writing to the outputstream
config	Page	same as ServletConfig
page	Page	this page's Servlet
exception	Page	exception created on this page.

Note: Care should be taken not to name your objects the same name as the implicit objects. If you have your own object with the same name, then the implicit objects take precedence over your own object.

Q 33: Explain hidden and output comments? SF

A 33: An output comment is a comment that is sent to the client where it is viewable in the browser's source. CO

```
<!--This is a comment which is sent to the client-->
```

A hidden comment documents a JSP page but does not get sent to the client. The JSP engine ignores a hidden comment, and does not process any code within hidden comment tags.

```
<%-- This comment will not be visible to the client --%>
```

Q 34: Is JSP variable declaration thread safe? CI

A 34: No. The declaration of variables in JSP is not thread-safe, because the declared variables end up in the generated Servlet as an instance variable, not within the body of the _jspservice() method.

The following declaration is not thread safe: because these are declarations, and will only be evaluated once when the page is loaded

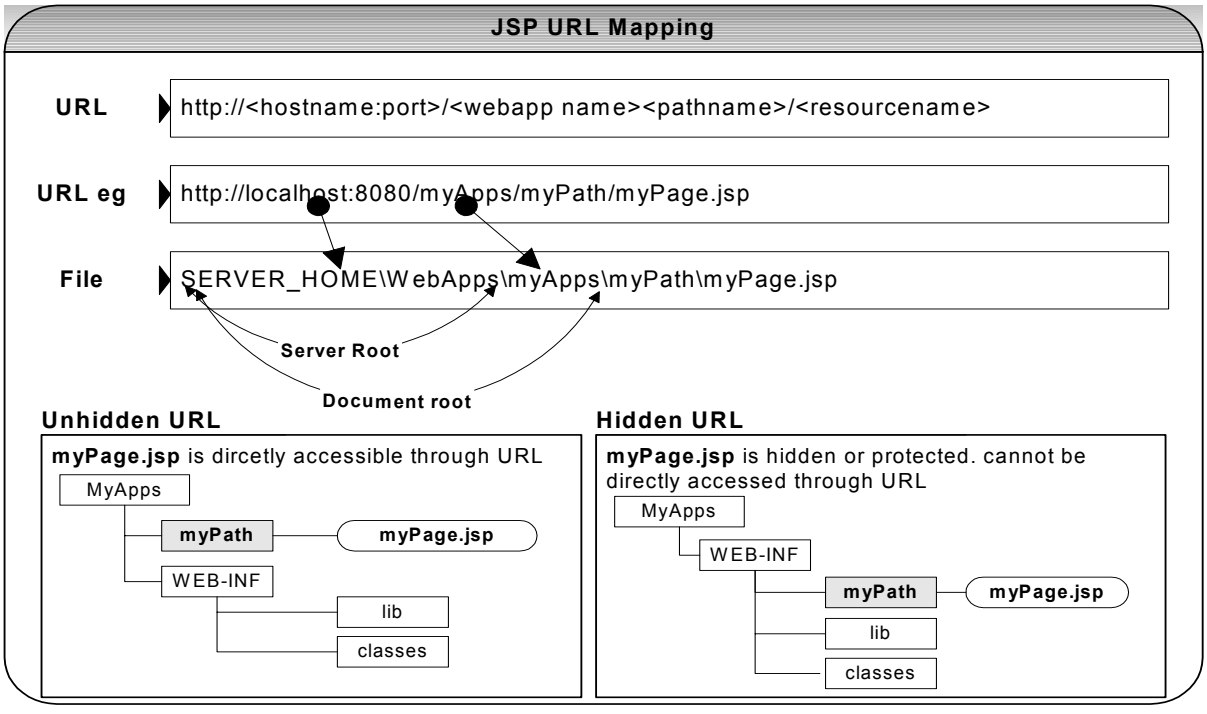
```
<%! int a = 5 %>
```

The following declaration is thread safe: because the variables declared inside the scriplets have the local scope and not shared.

```
<% int a = 5 %>
```

Q 35: Explain JSP URL mapping? What is URL hiding or protecting the JSP page? SF SE

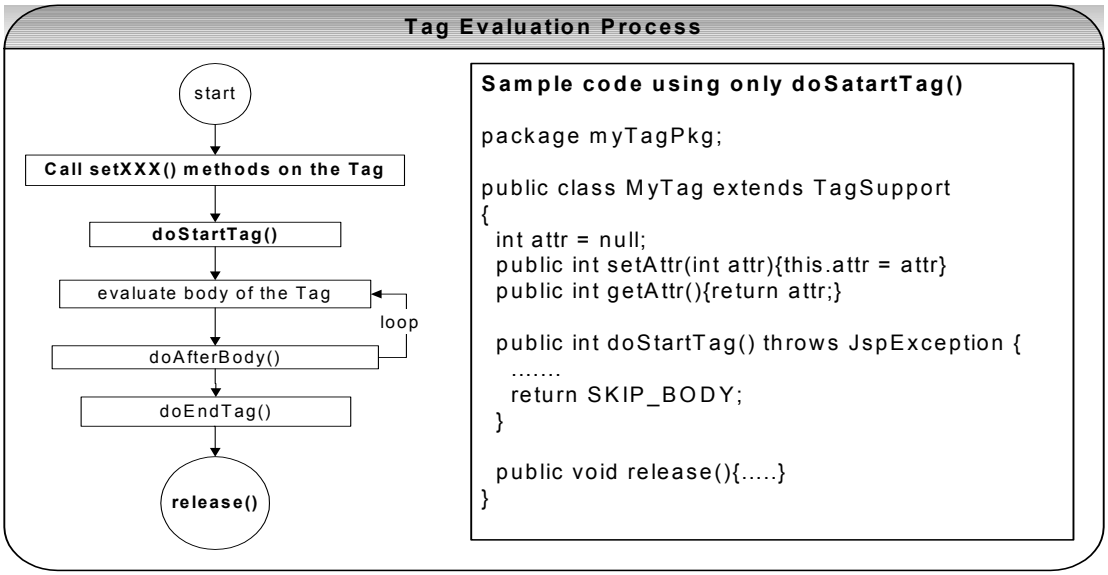
A 35: As shown in the figure, the JSP resources usually reside directly or under subdirectories (e.g. myPath) of the **document root,** which **are directly accessible** to the user through the URL. If you want to protect your Web resources then hiding the JSP files behind the WEB-INF directory can protect the JSP files, css (cascading style sheets) files, Java Script files, pdf files, image files, html files etc from direct access. The request should be made to a servlet who is responsible for authenticating and authorising the user before returning the protected JSP page or its resources.



Q 36: What are custom tags? Explain how to build custom tags? SF

A 36: Custom JSP tag is a tag you define. You define how a tag, its attributes and its body are interpreted, and then group your tags into collections called tag libraries that can be used in any number of JSP files. So basically it is a reusable and extensible JSP only solution. The pre-built tags also can speed up Web development. CO

STEP: 1
Construct the Tag handler class that defines the behaviour.



STEP: 2
The Tag library descriptor file (*.tld) maps the XML element names to the tag implementations. The code sample ***MyTagDesc.tld*** is shown below:

```
<taglib>
 <tag>
   <name>tag1</name>
   <tagclass>myTagPkg.MyTag</tagclass>
   <bodycontent>empty</bodycontent>
   <attribute>
     <name>attr</name>
     <required>false</required>
     <rtexprvalue>false</rtexprvalue>
   </attribute>
 </tag>
</taglib>
```

STEP: 3
The web.xml deployment descriptor maps the URI to the location of the *.tld (Tag Library Descriptor) file. The code sample web.xml file is shown below:

```
<web-app>
  <taglib>
    <taglib-uri>/WEB-INF/MyTagURI</taglib-uri>
    <taglib-location>/WEB-INF/tags/MyTagDesc.tld</taglib-location>
  </taglib>
</web-app>
```

STEP: 4
The JSP file declares and then uses the tag library as shown below:

```
<%@ taglib uri="/WEB-INF/ MyTagURI" prefix="myTag" %>

< myTag:tag1 attr=”abc”></ myTag:tag1> or < myTag:tag1 attr=”abc” />
```

Q 37: What is a TagExtraInfo class? **SF**
A 37: A TagExtraInfo class provides extra information about tag attributes to the JSP container at translation time.

- **Returns information about the scripting variables** that the tag makes available to the rest of the JSP page to use. The method used is:

```
VariableInfo[] getVariableInfo(TagData td)
```

Example

```
<html>
        <myTag:addObjectsToArray name="myArray" />
        <myTag:displayArray name="myArray" />
</html>
```

Without the use of TagExtraInfo, if you want to manipulate the attribute *myArray* in the above code in a scriptlet it will not be possible. This is because it does not place the *myArray* object on the page. You can still use pageContext.getAttribute() but that may not be a cleaner approach because it relies on the page designer to correctly cast to object type. The *TagExtraInfo* can be used to make items stored in the pageContext via setAttribute() method available to the scriptlet as shown below.

```
<html>
        <myTag:addObjectsToArray name="myArray" />
       <%-- scriplet code %>
        <% for(int i=0; i<myArray.length;i++){
                html +=  <LI> + myArray[i] + </LI>;
        %>
</html>
```

- **Validates the attributes passed to the Tag at translation time**.

Example It can validate the *myArray* array list to have not more than 100 objects. The method used is:

```
boolean isValid(TagData data)
```

Q 38: What is the difference between custom JSP tags and JavaBeans? **SF**
A 38: In the context of a JSP page, both accomplish similar goals but the differences are:

Custom Tags	JavaBeans
Can manipulate JSP content.	Can't manipulate JSP content.
Custom tags can simplify the complex operations much better than the bean can. But require a bit more work to set up.	Easier to set up.
Used only in JSPs in a relatively self-contained manner.	Can be used in both Servlets and JSPs. You can define a bean in one Servlet and use them in another Servlet or a JSP page.

JavaBeans declaration and usage example: CO

```
<jsp:useBean id="identifier" class="packageName.className"/>
<jsp:setProperty name="identifier" property="classField" value="someValue" />
<jsp:getProperty name="identifier" property="classField" /> <%=identifier.getclassField() %>
```

Q 39: Tell me about JSP best practices? **BP**
A 39:

- **Separate HTML code from the Java code**: Combining HTML and Java code in the same source code can make the code less readable. Mixing HTML and scriplet will make the code extremely difficult to read and maintain. The display or behaviour logic can be implemented as a custom tags by the Java developers and Web designers can use these Tags as the ordinary XHTML tags.

- **Place data access logic in JavaBeans**: The code within the JavaBean is readily accessible to other JSPs and Servlets.

- **Factor shared behaviour out of Custom Tags into common JavaBeans classes**: The custom tags are not used outside JSPs. To avoid duplication of behaviour or business logic, move the logic into JavaBeans and get the custom tags to utilize the beans.

- **Choose the right "*include"* mechanism:** What are the differences between static and a dynamic include? Using includes will improve code reuse and maintenance through modular design. Which one to use? Refer **Q31** in Enterprise section.

- **Use style sheets** (e.g. css)**, template mechanism** (e.g. struts tiles etc) and **appropriate comments** (both hidden and output comments)**.**

Q 40: How will you avoid scriptlet code in JSP? **BP**
A 40: Use JavaBeans or Custom Tags instead.

Enterprise - JDBC

Q 41: What is JDBC? How do you connect to a database? **SF**
A 41: JDBC stands for **Java Database Connectivity**. It is an API which provides easy connection to a wide range of databases. To connect to a database we need to load the appropriate driver and then request for a connection object. The Class.forName(….) will load the driver and register it with the DriverManager (Refer **Q4** in Java section for dynamic class loading).

```
Class.forName("oracle.jdbc.driver.OracleDriver");
String url = jdbc:oracle:thin:@hostname:1526:myDB;
Connection myConnection =  DriverManager.getConnection(url, "username", "password");
```

The ***DataSource*** interface provides an alternative to the *DriverManager* for making a connection. *DataSource* makes the code more portable than DriverManager because it work with JNDI and it is created, deployed and managed separately from the application that uses it. If the DataSource location changes, then there is no need to change the code but change the configuration properties in the server. This makes your application code easier to maintain. *DataSource* allows the use of connection pooling and support for distributed transactions. A *DataSource* is not only a database but also can be a file or a spreadsheet. A *DataSource* object can be bound to JNDI and an application can retrieve and use it to make a connection to the database. J2EE application servers provide tools to define your DataSource with a JNDI name. When the server starts it loads all the DataSources into the Application Server's JNDI service.

DataSource configuration properties are shown below:

- **JNDI Name** → jdbc/myDataSource
- **URL** → jdbc:oracle:thin:@hostname:1526:myDB
- **UserName**, **Password**
- **Implementation classname** → oracle.jdbc.pool.OracleConnectionPoolDataSource
- **Classpath** → ora_jdbc.jar
- **Connection pooling** settings like → minimum pool size, maximum pool size, connection timeout, statement cache size etc.

Once the *DataSource* has been set up, then you can get the connection object as follows:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/myDataSource");
Connection myConnection = ds.getConnection("username","password");
```

In a basic implementation a *Connection* obtained from a *DataSource* and a *DriverManager* are identical. But, **DataSource is recommended because of its better portability**.

Design Pattern: JDBC architecture decouples an abstraction from its implementation so that the implementation can vary independent of the abstraction. This is an example of the **bridge design pattern**. The JDBC API provides the abstraction and the JDBC drivers provide the implementation. New drivers can be plugged-in to the JDBC API without changing the client code.

Q 42: What are JDBC Statements? What are different types of statements? How can you create them? **SF**

A 42: A **statement** object is responsible for sending the SQL statements to the Database. Statement objects are created from the connection object and then executed. **CO**

```
Statement stmt = myConnection.createStatement();
ResultSet rs = stmt.executeQuery("SELECT id, name FROM myTable where id =1245");// to read
                                                    Or
stmt.executeUpdate("INSERT INTO (field1,field2) values (1,3)");// to insert/update/delete/create table
```

The types of statements are:

- **Statement** (regular statement as shown above)
- **PreparedStatement** (more efficient than statement due to pre-compilation of SQL)
- **CallableStatement** (to call stored procedures on the database)

To use prepared statement:

```
PreparedStatement prepStmt =
        myConnection.prepareStatement("SELECT id, name FROM myTable where id = ? ");
prepStmt.setInt(1, 1245);
```

Callable statements are used for calling stored procedures.

```
CallableStatement calStmt = myConnection.prepareCall("{call PROC_SHOWMYBOOKS}");
ResultSet rs = cs.executeQuery();
```

Q 43: What is a **Transaction**? What does **setAutoCommit** do? **TI** **PI**

A 43: A transaction is a set of operations that should be completed as a unit. If one operation fails then all the other operations fail as well. For example if you transfer funds between two accounts there will be two operations in the set

1. Withdraw money from one account.
2. Deposit money into other account.

These two operations should be completed as a single unit. Otherwise your money will get lost if the withdrawal is successful and the deposit fails. There are four characteristics (**ACID** properties) for a Transaction.

Atomicity	**C**onsistency	**I**solation	**D**urability
All the individual operations should either complete or fail.	The design of the transaction should update the database correctly.	Prevents data being corrupted by concurrent access by two different sources. It keeps transactions isolated or separated from each other until they are finished.	Ensures that the database is definitely updated once the Transaction is completed.

Transactions maintain data integrity. A transaction has a beginning and an end like everything else in life. The **setAutocommit(….)**, **commit()** and **rollback()** are used for marking the transactions (known as transaction demarcation). When a connection is created, it is in **auto-commit** mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed immediately after it is executed. The way to allow two or more statements to be grouped into a transaction is to **disable** auto-commit mode: **CO**

```
try{
   Connection myConnection = dataSource.getConnection();

   // set autoCommit to false
   myConnection .setAutoCommit(false);

    withdrawMoneyFromFirstAccount(.....);   //operation 1
    depositMoneyIntoSecondAccount(.....);   //operation 2

   myConnection .commit();
}
catch(Exception sqle){
   try{
      myConnection .rollback();
   }catch( Exception e){}
}
finally{
   try{if( conn != null) {conn.close();}} catch( Exception e) {}
}
```

The above code ensures that both operation 1 and operation 2 succeed or fail as an atomic unit and consequently leaves the database in a consistent state. Also turning auto-commit off will provide better performance.

Q 44: What is the difference between JDBC-1.0 and JDBC-2.0? What are Scrollable ResultSets, Updateable ResultSets, RowSets, and Batch updates? **SF**

A 44: JDBC2.0 has the following additional features or functionality:

JDBC 1.0	JDBC 2.0
With JDBC-1.0 the ResultSet functionality was limited. There was no support for updates of any kind and scrolling through the ResultSets was forward only (no going back)	With JDBC 2.0 ResultSets are updateable and also you can move forward and backward. **Example** This example creates an updateable and scroll-sensitive ResultSet `Statement stmt = myConnection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATEABLE)`

With JDBC-1.0 the statement objects submits updates to the database individually within same or separate transactions. This is very inefficient large amounts of data need to be updated.	With JDBC-2.0 statement objects can be grouped into a batch and executed at once. We call addBatch() multiple times to create our batch and then we call executeBatch() to send the SQL statements off to database to be executed as a batch (this minimises the network overhead).
	
	Example
	
	`Statement stmt = myConnection.createStatement();`
	`stmt.addBatch("INSERT INTO myTable1 VALUES (1,"ABC")");`
	`stmt.addBatch("INSERT INTO myTable1 VALUES (2,"DEF")");`
	`stmt.addBatch("INSERT INTO myTable1 VALUES (3,"XYZ")");`
	`…`
	`int[] countInserts = stmt.executeBatch();`

-	The JDBC-2.0 optional package provides a RowSet interface, which extends the ResultSet. One of the implementations of the RowSet is the CachedRowSet, which can be considered as a disconnected ResultSet.

Q 45: How to avoid the "running out of cursors" problem? **DC** **PI** **MI**

A 45: A database can run out of cursors if the connection is not closed properly or the DBA has not allocated enough cursors. In a Java code it is essential that we close all the valuable resources in a try{} and **finally**{} block. The finally{} block is always executed even if there is an exception thrown from the catch {} block. So the resources like connections and statements should be closed in a finally {} block. **CO**

Try{} Finally {} blocks to close Exceptions

Wrong Approach -

Connections and statements will not be closed if there is an exception:

```
public void executeSQL() throws SQLException{

    Connection con =   DriverManager.getConnection(.....);
     .....
    Statement stmt = con.createStatement();
    ....
    //line 20  where exception is thrown
    ResultSet rs = stmt.executeQuery("SELECT * from myTable");
     .....
    rs.close();
    stmt.close();
    con.close();
}
```

Note: if an exception is thrown at line 20 then the close() statements are never reached.

Right Approach -

```
public void executeSQL() throws SQLException{
   try{
     Connection con =   DriverManager.getConnection(.....);
      .....
     Statement stmt = con.createStatement();
     ....
     //line 20  where exception is thrown
     ResultSet rs = stmt.executeQuery("SELECT * from myTable");
      .....
   }
   finally{
       try {
          if(rs != null) rs.close();
          if(stmt != null) stmt.close();
          if(con != null) con.close();
       }
        catch(Exception e){}
   }
}
```

Note: if an exception is thrown at line 20 then the finally clause is called before the exception is thrown to the method.

Q 46: What is the difference between statements and prepared statements? SF PI SE BP

A 46:

- Prepared statements offer better performance, as they are **pre-compiled**. Prepared statements reuse the same **execution plan** for different arguments rather than creating a new execution plan every time. Prepared statements use bind arguments, which are sent to the database engine. This allows mapping different requests with same prepared statement but different arguments to execute the same execution plan.

- Prepared statements are more secure because they use bind variables, which can prevent SQL injection attack.

  The most common type of SQL injection attack is SQL manipulation. The attacker attempts to modify the SQL statement by adding elements to the WHERE clause or extending the SQL with the set operators like UNION, INTERSECT etc.

  Example Let us look at the following SQL:

```
SELECT * FROM users where username='bob' AND password='xyfdsw' ;
```

  The attacker can manipulate the SQL as follows

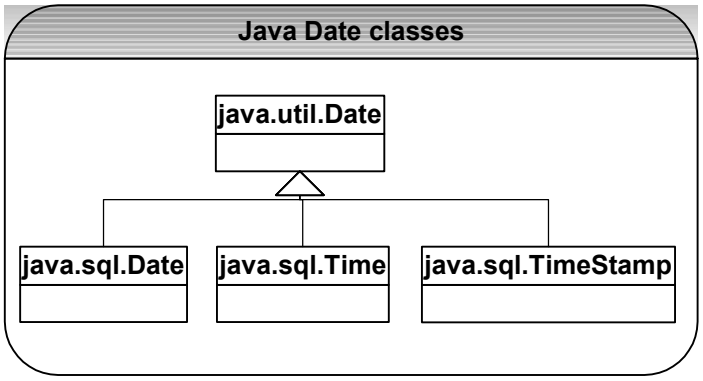
```
SELECT * FROM users where username='bob' AND password='xyfdsw' OR 'a' = 'a' ;
```

  The above "WHERE" clause is always true because of the operator precedence. The PreparedStatement can prevent this by using bind variables:

```
String strSQL = SELECT * FROM users where username=? AND password=?);
PreparedStatement pstmt = myConnection.prepareStatement(strSQL);
pstmt.setString(1,"bob");
pstmt.setString(2, "xyfdsw");
pstmt.execute();
```

Q 47: Explain differences among java.util.Date, java.sql.Date, java.sql.Time, and java.sql.Timestamp? SF

A 47: As shown below all the sql Date classes extend the util Date class.



java.util.Date - class supports both the Date (ie year/month/date etc) and the Time (hour, minute, second, and millisecond) components.

java.sql.Date - class supports only the Date (ie year/month/date etc) component. The hours, minutes, seconds and milliseconds of the Time component will be set to zero in the particular time zone with which the instance is associated.

java.sql.Time - class supports only Time (ie hour, minute, second, and millisecond) component. The date components should be set to the "zero epoch" value of January 1, 1970 and should not be accessed.

java.sql.TimeStamp – class supports both Date (ie year/month/date etc) and the Time (hour, minute, second, millisecond and **nanosecond**) components.

Note: the subtle difference between **java.util.Date** and **java.sql.Date.**

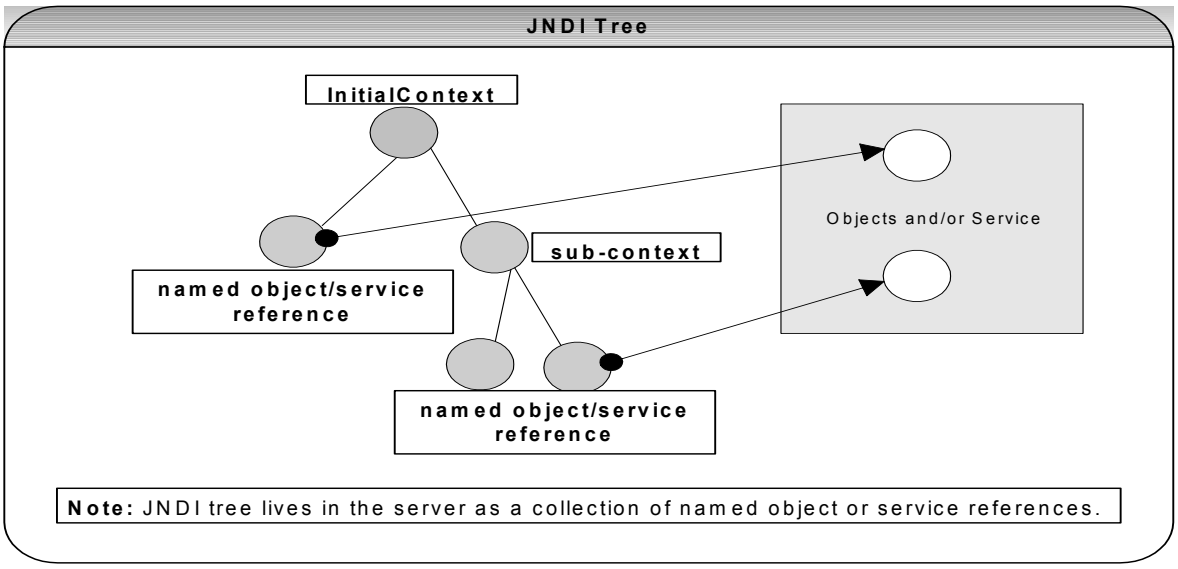
To keep track of time Java counts the number of milliseconds from January 1, 1970 and stores it as a long value in **java.util.Date** class. The **GregorianCalendar** class provides us a way to represent an arbitrary date. The **GregorianCalendar** class also provides methods for manipulating dates.

Enterprise – JNDI & LDAP

Q 48: What is JNDI? And what are the typical uses within a J2EE application? **SF**

A 48: JNDI stands for **J**ava **N**aming and **D**irectory **I**nterface. It provides a generic interface to LDAP (**L**ightweight **D**irectory **A**ccess **P**rotocol) and other directory services like NDS, DNS (Domain Name System) etc. It provides a means for an application to locate components that exist in a name space according to certain attributes. A J2EE application component uses JNDI interfaces to look up and reference system-provided and user-defined objects in a component environment. JNDI is not specific to a particular naming or directory service. It can be used to access many different kinds of systems including file systems.

The JNDI API enables applications to look up objects such as DataSources, EJBs, MailSessions and JMS by name. The Objects can be loaded into the JNDI tree using a J2EE application server's administration console. To load an object in a JNDI tree, choose a name under which you want the object to appear in a JNDI tree. J2EE deployment descriptors indicate the placement of J2EE components in a JNDI tree.



The parameters you have to define for JNDI service are as follows:

- The name service provider class name (WsnInitialContext for Websphere).

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.ibm.websphere.naming.WsnInitialContextFactory");
```

- The provider URL :
    - The name service hostname.
    - The name service port number.

```
env.put(Context. PROVIDER_URL, " iiop://localhost:1050");
Context ctx = new InitialContext(env);
```

JNDI is **like** a file system or a Database.

File System	JNDI	Database
File system starts with a **mounted drive** like c:\	JNDI starts with an **InitialContext**. i.e. new InitialContext().	**Database instance**

Uses a **subdirectory**. C:\subdir1	Navigate to a **sub-context**. e.g. Subcontext1	**Tablespace**
Access a **subdirectory** c:\subdir1\subdir2	Drill down through other **sub-contexts**. e.g. subcontext1/subcontext2	**Table**
Access a **file**. C:\subdir1\subdir2\myFile	Access an **object** or a **service**. New InitialContext().lookup("objectName");	**Data**
Example: c:\subdir1\subdir2\myFile	**Example:** iiop://myserver:2578/subcontext1.subcontext2.objectName	**Example:** Select * from demo.myTable

---

Q 49: Explain the difference between the look up of "java:comp/env/ejb/MyBean" and "ejb/MyBean"? SF

A 49:

java:comp/env/ejb/MyBean	ejb/MyBean
This is a logical reference, which will be used in your code.	This is a physical reference where an object will be mapped to in a JNDI tree.

The logical reference (or alias) **java:comp/env/ejb/MyBean** is the recommended approach because you cannot guarantee that the physical JNDI location (**ejb/MyBean**) you specify in your code will be available. Your code will break if the physical location is changed. The deployer will not be able to modify your code. Logical references solve this problem by binding the logical name to the physical name in the application server. The logical names will be declared in the deployment descriptors (web.xml and/or ejb-jar.xml) as follows and these will be mapped to physical JNDI locations in the application server specific deployment descriptors.

To look up a JDBC resource from either WEB (web.xml) or EJB (ejb-jar.xml) tier, the deployment descriptor should have the following entry:

```
<resource-ref>
    <description>The DataSource</description>
    <res-ref-name>jdbc/MyDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

This will make full logical path to the bean as:
java:comp/env/jdbc/MyDataSource

To use it:

```
Context ctx = new InitialContext();
Object ref = ctx.lookup(java:comp/env/jdbc/MyDataSource);
```

To look up EJBs from another EJB or a WEB module, the deployment descriptor should have the following entry:

```
<ejb-ref>
    <description>myBean</description>
    <ejb-ref-name>ejb/MyBean</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <ejb-link>Region</ejb-link>
    <home>com.MyBeanHome</home>
    <remote>com.MyBean</remote>
</ejb-ref>
```

This will make full logical path to the bean as:
java:comp/env/ejb/MyBean

To use it:

```
Context ctx = new InitialContext();
Object ref = ctx.lookup(java:comp/env/ejb/MyBean);
```

Q 50: What is a JNDI InitialContext? SF

A 50: All naming operations are relative to a context. The *InitalContext* implements the *Context* interface and provides an **entry point** for the resolution of names.

Q 51: What is an LDAP server? And what is it used for in an enterprise environment? SF SE

A 51: LDAP stands for **L**ightweight **D**irectory **A**ccess **P**rotocol. This is an extensible open network protocol standard that provides access to distributed directory services. LDAP is an Internet standard for directory services that run on TCP/IP. Under OpenLDAP and related servers, there are two servers – **slapd**, the LDAP daemon where the queries are sent to and **slurpd**, the replication daemon where data from one server is pushed to one or more

slave servers. By having multiple servers hosting the same data, you can increase reliability, scalability, and availability.

- It defines the operations one may perform like search, add, delete, modify, change name
- It defines how operations and data are conveyed.

LDAP has the potential to consolidate all the existing application specific information like user, company phone and e-mail lists. This means that the change made on an LDAP server will take effect on every directory service based application that uses this piece of user information. The variety of information about a new user can be added through a single interface which will be made available to Unix account, NT account, e-mail server, Web Server, Job specific news groups etc. When the user leaves his account can be disabled to all the services in a single operation.

So LDAP is most useful to provide “white pages” (e.g. names, phone numbers, roles etc) and “yellow pages” (e.g. location of printers, application servers etc) like services. Typically in a J2EE application environment it will be used to authenticate and authorise users.

Why use LDAP when you can do the same with relational database (RDBMS)?

In general LDAP servers and RDBMS are designed to provide different types of services. LDAP is an open standard access mechanism, so an RDBMS can talk LDAP. However the servers, which are built on LDAP, are **optimized for read access** so likely to be much faster than RDBMS in providing read access. So in a nutshell, **LDAP is more useful when the information is often searched but rarely modified**. (Another difference is that RDBMS systems store information in rows of tables whereas LDAP uses object oriented hierarchies of entries.) .

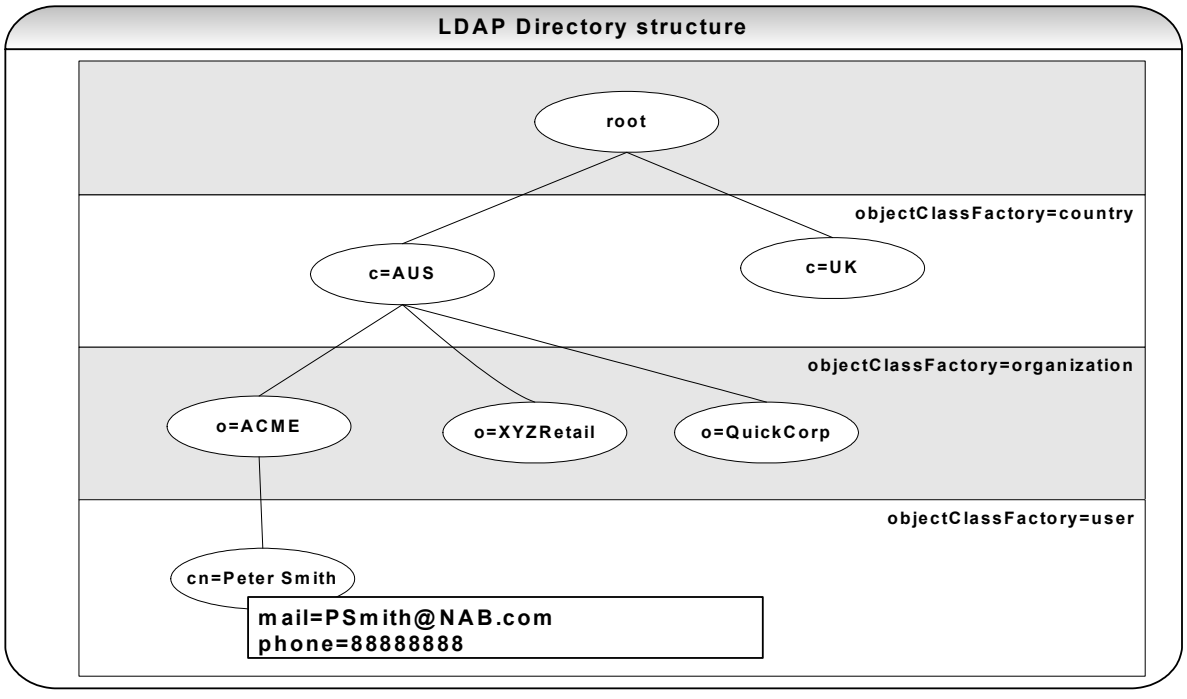
Key LDAP Terms:

DIT: **D**irectory **I**nformation **T**ree. Hierarchical structure of entries, those make up a directory.

DN: **D**istinguished **N**ame. This uniquely identifies an entry in the directory. A **DN is made up of relative DN**s of the entry and each of entry’s parent entries up to the root of the tree. DN is read from right to left and commas separate these names. For example ‘**cn=**Peter Smith, **o=**ACME, **c=**AUS’.

objectClass: An *objectClass* is a formal definition of a specific kind of objects that can be stored in the directory. An ObjectClass is a distinct, named set of attributes that represent something concrete such as a user, a computer, or an application.

LDAP URL: This is a string that specifies the location of an LDAP resource. An LDAP URL consists of a server host and a port, search scope, **baseDN**, filter, attributes and extensions. Refer to diagram below:



So the complete distinguished name for bottom left entry (ie Peter Smith) is **cn**=Peter Smith, **o**=ACME, **c**=AUS. Each entry must have at least one attribute that is used to name the entry. To manage the part of the LDAP directory we should specify the highest level parent distinguished names in the server configuration. These distinguished names are called **suffixes**. The server can access all the objects that are below the specified suffix in the hierarchy. For example in the above diagram to answer queries about 'Peter Smith' the server should have the **suffix** of 'o=ACME, **c**=AUS'. So we can look for "Peter Smith" by using the following distinguished name:

```
cn=Peter Smith, o=ACME, c=AUS   //where o=ACME, c=AUS is the suffix
```

LDAP schema: defines rules that specify the types of objects that a directory may contain and the required optional attributes that entries of different types should have.

Filters: In LDAP the basic way to retrieve data is done with filters. There is a wide variety of operators that can be used as follows: & (and), | (or), ! (not), ~= (approx equal), >= (greater than or equal), <= (less than or equal), * (any) etc.

```
(& (uid=a*) (uid=*l) )
```

So where does JNDI fit into this LDAP? JNDI provides a standard API for interacting with naming and directory services using a service provider interface (SPI), which is analogous to JDBC driver. To connect to an LDAP server, you must obtain a reference to an object that implements the ***DirContext***. In most applications, this is done by using an *IntialDirContext* object that takes a Hashtable as an argument:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:387");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=Directory Manager");
env.put(Context.SECURITY_CREDENTIALS, "myPassword");
DirContext ctx = new InitialDirContext(env);
```

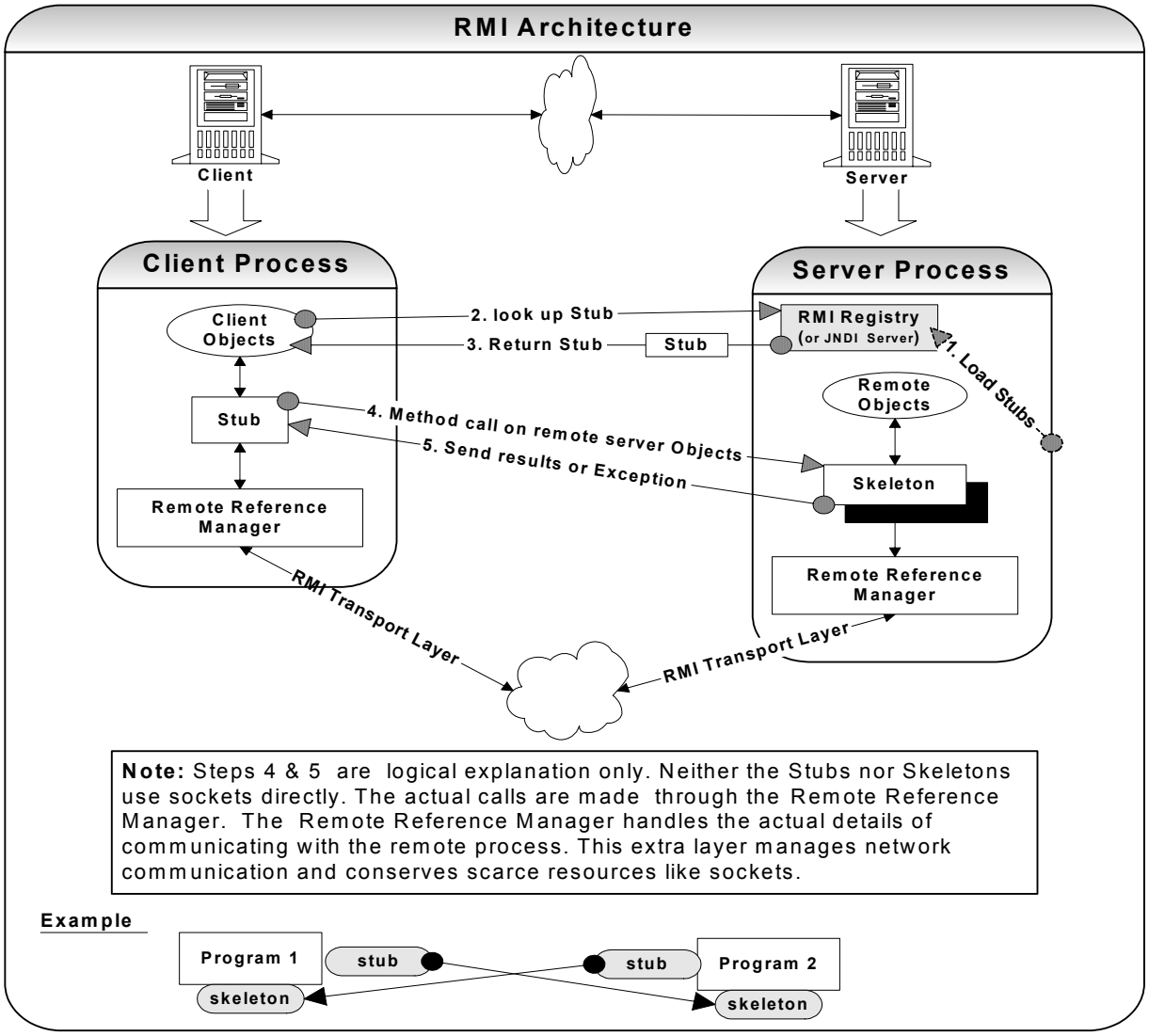
Enterprise - RMI

Q 52: Explain the RMI architecture? SF

A 52: Java Remote Method Invocation (RMI) provides a way for a Java program on one machine to communicate with objects residing in different JVMs (or processes or address spaces). The important parts of the RMI architecture are the stub class, object serialization and the skeleton class. RMI uses a layered architecture where each of the layers can be enhanced without affecting the other layers. The layers can be summarised as follows:

- **Application Layer:** The client and server program
- **Stub & Skeleton Layer:** Intercepts method calls made by the client. Redirects these calls to a remote RMI service.
- **Remote Reference Layer**: Sets up connections to remote address spaces, manages connections, and understands how to interpret and manage references made from clients to the remote service objects.
- **Transport layer:** Based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.

Design pattern: RMI stub classes provide a reference to a skeleton object located in a different address space on the same or different machine. This is a typical example of a **proxy design pattern** (i.e. remote proxy), which makes an object executing in another JVM appear like a local object. In JDK 5.0 and later, the RMI facility uses **dynamic proxies** instead of generated stubs, which makes RMI easier to use. Refer **Q11** in "How would you about…" section for a more detailed discussion on proxy design pattern and dynamic proxies.



RMI runtime steps (as shown in the diagram above) involved are:

Step 1: Start RMI registry and then the RMI server. Bind the remote objects to the RMI registry.
Step 2: The client process will look up the remote object from the RMI registry.
Step 3: The lookup will return the stub to the client process from the server process.
Step 4: The client process will invoke method calls on the stub. The stub calls the skeleton on the server process through the RMI reference manager.
Step 5: The skeleton will execute the actual method call on the remote object and return the result or an exception to the client process via the RMI reference manager and the stub.

Q 53: What is a remote object? Why should we extend ***UnicastRemoteObject***? SF

A 53: A remote object is one whose methods can be invoked from another JVM (or process). A remote object class must implement the *Remote* interface. A RMI Server is an application that creates a number of remote objects.

An RMI Server is responsible for

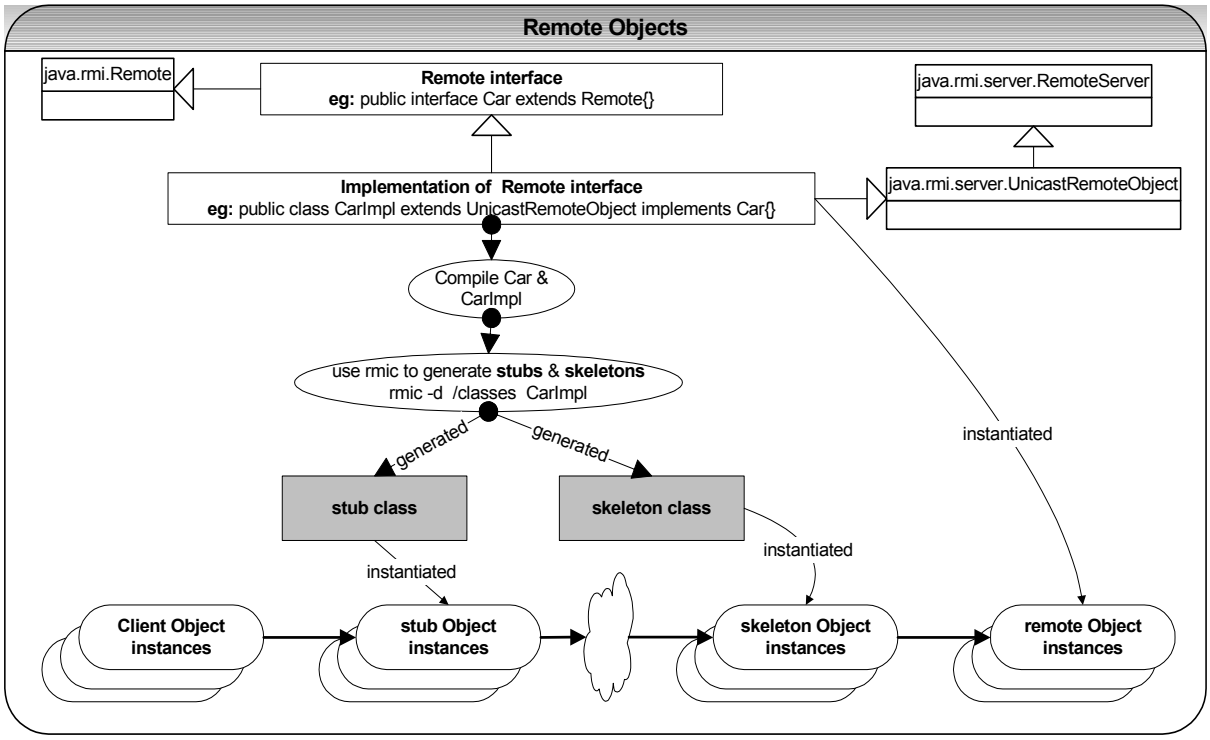
- Creating an instance of the remote object (e.g. CarImpl instance = new CarImpl()).
- **Exporting** the remote object.
- Binding the instance of the remote object to the RMI registry.

By exporting a remote object you make it available to accept incoming calls from the client. You can export the remote object by either extending the java.rmi.server.UnicastRemoteObject or if your class is already extending another class then you can use the static method

```
UnicastRemoteObject.exportObject (this);
```

If the UnicastRemoteObject is not extended (ie if you use UnicastRemoteObject.exportObject(…) then the implementation class is responsible for the correct implementations of the hashCode(), equals() and toString() methods. A remote object is registered in the RMI registry using:

```
Naming.rebind(String serviceName, Remote remoteObj);
```



Q 54: What is the difference between RMI and CORBA? SF

A 54:

RMI	CORBA
Java only solution. The interfaces, implementations and the clients are all written in Java.	CORBA was made specifically for interoperability among various languages. For example the server could be written in C++ and the business logic can be in Java and the client can be written in COBOL.
RMI allows dynamic loading of classes at runtime.	In a CORBA environment with multi-language support it is not possible to have dynamic loading.

Q 55: What are the services provided by the RMI Object? SF

A 55: In addition to its remote object architecture RMI provides some basic object services, which can be used in a distributed application. These services are

- **Object naming/registry service**: RMI servers can provide services to clients by registering one or more remote objects with its local RMI registry.
- **Object activation service**: It provides a way for server (or remote) objects to be started on an as-needed basis. Without the remote activation service, a server object has to be registered with the RMI registry service.
- **Distributed garbage collection:** It is an automatic process where an object, which has no further remote references, becomes a candidate for garbage collection.

Q 56: What are the differences between RMI and a socket? SF

A 56:

Socket	RMI
A socket is a transport mechanism. Sockets are like applying procedural networking to object oriented environment.	RMI uses sockets. RMI is object oriented. Methods can be invoked on the remote objects running on a separate JVM.
Sockets-based network programming can be laborious.	RMI provides a convenient abstraction over raw sockets. Can send and receive any valid Java object utilizing underlying object serialization without having to worry about using data streams.

Q 57: How will you pass parameters in RMI? SF

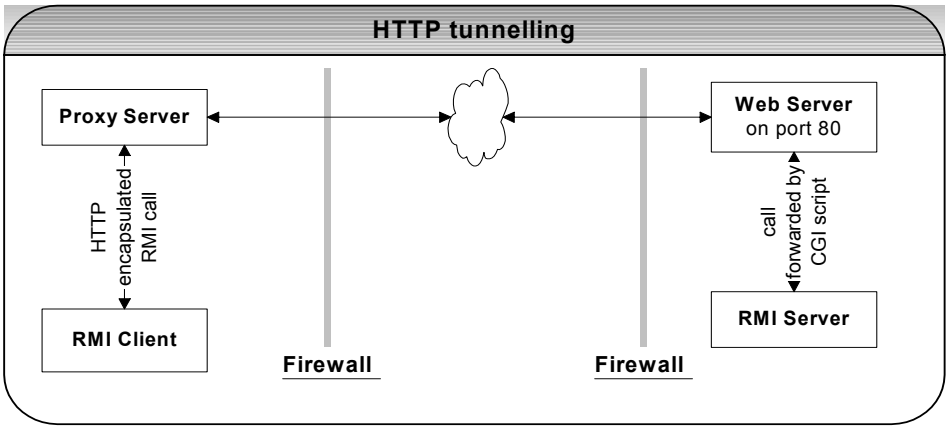
A 57:

- Primitive types are passed by value (e.g. int, char, boolean etc).
- References to remote objects (i.e. **objects which implements the Remote interface**) are passed as remote references that allows the client process to invoke methods on the remote objects.
- Non-remote objects are passed by value using object serialization. These objects should allow them to be serialized by implementing the java.io.*Serializable* interface.

Note: The client process initiates the invocation of the remote method by calling the method on the stub. The stub (client side proxy of the remote object) has a reference to the remote object and forwards the call to the skeleton (server side proxy of the remote object) through the reference manager by **marshalling** the method arguments. During marshalling each object is checked to determine whether it implements *java.rmi.Remote* interface. If it does then the remote reference is used as the marshalled data otherwise the object is serialized into byte streams and sent to the remote process where it is deserialized into a copy of the local object. The skeleton converts this request from the stub into the appropriate method call on the actual remote object by **unmarshalling** the method arguments into local stubs on the server (if they are remote reference) or into local copy (if they are sent as serialized objects).

Q 58: What is HTTP tunnelling or how do you make RMI calls across firewalls? SF SE

A 58: RMI transport layer generally opens direct sockets to the server. Many Intranets have firewalls that do not allow this. To get through the firewall an RMI call can be embedded within the firewall-trusted HTTP protocol. To get across firewalls, RMI makes use of **HTTP tunnelling** by encapsulating RMI calls within an HTTP POST request.



When a firewall proxy server can forward HTTP requests only to a well-known HTTP port: The firewall proxy server will forward the request to a HTTP server listening on port 80, and a CGI script will be executed to forward the call to the target RMI server port on the same machine.

When a firewall proxy server can forward HTTP requests to any arbitrary port: The firewall proxy will forward to any arbitrary port on the host machine and then it is forwarded directly to the port on which RMI Server is listening.

The disadvantages of HTTP tunnelling are performance degradation, prevents RMI applications from using call-backs, CGI script will redirect any incoming request to any port, which is a security loophole, RMI calls cannot be multiplexed through a single connection since HTTP tunnelling follows a request/response protocol etc.

Q 59: Why use RMI when we can achieve the same benefits from EJB? **SF**

A 59: EJBs are distributed components, which use the RMI framework for object distribution. An EJB application server provides more services like transactions, object pooling, database connection-pooling etc, which RMI does not provide. These extra services that are provided by the EJB server simplify the programming effort at the cost of performance overhead compared to plain RMI. So if performance is important then pure RMI may be a better solution (or under extreme situations Sockets can offer better performance than RMI).

Note: The decision to go for RMI or EJB or Sockets should be based on requirements such as maintainability, ease of coding, extensibility, performance, scalability, availability of application servers, business requirements etc.

Enterprise – EJB 2.x

There are various persistence mechanisms available like EJB 2.x, Object-to-Relational (O/R) mapping tools like Hibernate, JDBC and EJB 3.0 (new kid on the block) etc. You will have to evaluate the products based on the application you are building because each product has its strengths and weaknesses. You will find yourself trading ease of use for scalability, standards with support for special features like stored procedures, etc. Some factors will be more important to you than for others. There is no one size fits all solution. Let's compare some of the persistence products:

EJB 2.x	EJB 3.0	Hibernate	JDBC
PROS: ▪ Security is provided for free for accessing the EJB. ▪ Provides declarative transactions. ▪ EJBs are pooled and cached. EJB life cycles are managed by the container. ▪ Has remote access capabilities and can be clustered for scalability.	**PROS:** ▪ A lot less artefacts than EJB 2.x. Make use of annotations or attributes based programming. ▪ Narrows the gap between EJB 2.x and O/R mapping. ▪ Do support OO concepts like inheritance.	**PROS:** ▪ Simple to write CRUD (create, retrieve, update, delete) operations. ▪ No container or application server is required and can be plugged into an existing container. ▪ Tools are available to simplify mapping relational data to objects and quick to develop.	**PROS:** ▪ You have complete control over the persistence because this is the building blocks of nearly all other persistence technologies in Java. ▪ Can call Stored Procedures. ▪ Can manipulate relatively large data sets.
Cons: ▪ Need to understand the intricacies like rolling back a transaction, granularity etc, infrastructures like session facades, business delegates, value objects etc and strategies like lazy loading, dirty marker etc. ▪ EJBs use lots of resources and have lots of artifacts. ▪ Does not support OO concepts like inheritance.	**Cons:** ▪ Since it is new, might be too early to use in commercial projects. ▪ It is still evolving.	**Cons:** ▪ Little or no capabilities for remote access and distributability. ▪ Mapping schemas can be tedious and O/R mapping has its tricks like using lazy initialization, eager loading etc. What works for one may not work for another. ▪ Limited clustering capabilities. ▪ Large data sets can still cause memory issues. ▪ Support for security at a database level only and no support for role based security without any add on APIs like Aspect Oriented Programming etc.	**Cons:** ▪ You will have to write a lot of code to perform a little. Easy to make mistakes in properly managing connections and can cause out of cursors issues. ▪ Harder to maintain because changes in schemas can cause lot of changes to your code. ▪ Records need to be locked manually (e.g. select for update).
As a rule of thumb, suitable for distributed and clustered applications, which is heavily transaction based. Records in use say between 1 and 50.	As a rule of thumb, suitable for distributed and clustered applications, which is heavily transaction based. Records in use say between 1 and 100.	Suitable for records in use between 100 and 5000. Watch out for memory issues, when using large data sets.	Where possible stay away from using JDBC unless you have compelling reason to use it for batch jobs where large amount of data need to be transferred, records in use greater than 5000, required to use Stored Procedures etc.

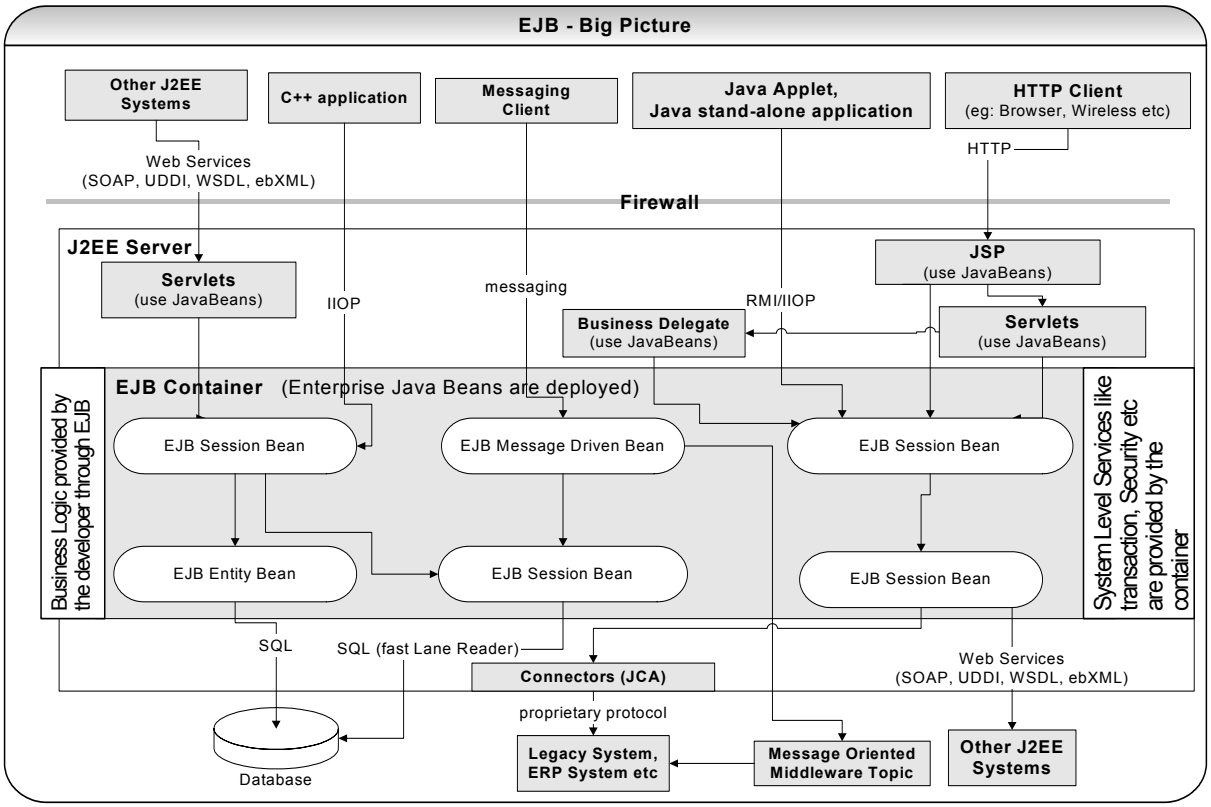
The stateless session beans and message driven beans have wider acceptance in EJB 2.x compared to stateful session beans and entity beans. Refer Emerging Technologies/Frameworks section for Hibernate and EJB 3.0.

Q 60: What is the role of EJB 2.x in J2EE? **SF**

A 60: EJB 2.x (Enterprise JavaBeans) is a widely adopted server side component architecture for J2EE.

- EJB is a remote, distributed multi-tier system and supports protocols like JRMP, IIOP, and HTTP etc.
- It enables rapid development of reusable, versatile, portable business components across middleware, transactional and scalable applications.
- EJB is a specification for J2EE servers. EJB components contain only business logic and system level programming and services like transactions, security, instance pooling, threading, persistence etc are managed by the EJB Container and hence simplify the programming effort.
- Message driven EJBs have support for asynchronous communication.

Note: Having said that EJB 2.x is a widely adopted server side component, **EJB 3.0** is taking ease of development very seriously and has adjusted its model to offer the POJO (Plain Old Java Object) persistence and the new **O/R mapping model based on Hibernate**. In EJB 3.0, **all kinds of enterprise beans are just POJOs**. EJB 3.0 **extensively uses Java annotations**, which replaces excessive XML, based configuration files and eliminates the need for the rigid component model used in EJB 1.x, 2.x. Annotations can be used to define the bean's business interface, O/R mapping information, resource references etc. Refer **Q18** in Emerging Technologies/Frameworks section. So, for future developments look out for EJB 3.0 and/or Hibernate framework. Refer **Q14 – Q16** in Emerging Technologies/Frameworks section for discussion on Hibernate framework.



Q 61: What is the difference between EJB and JavaBeans? **SF**

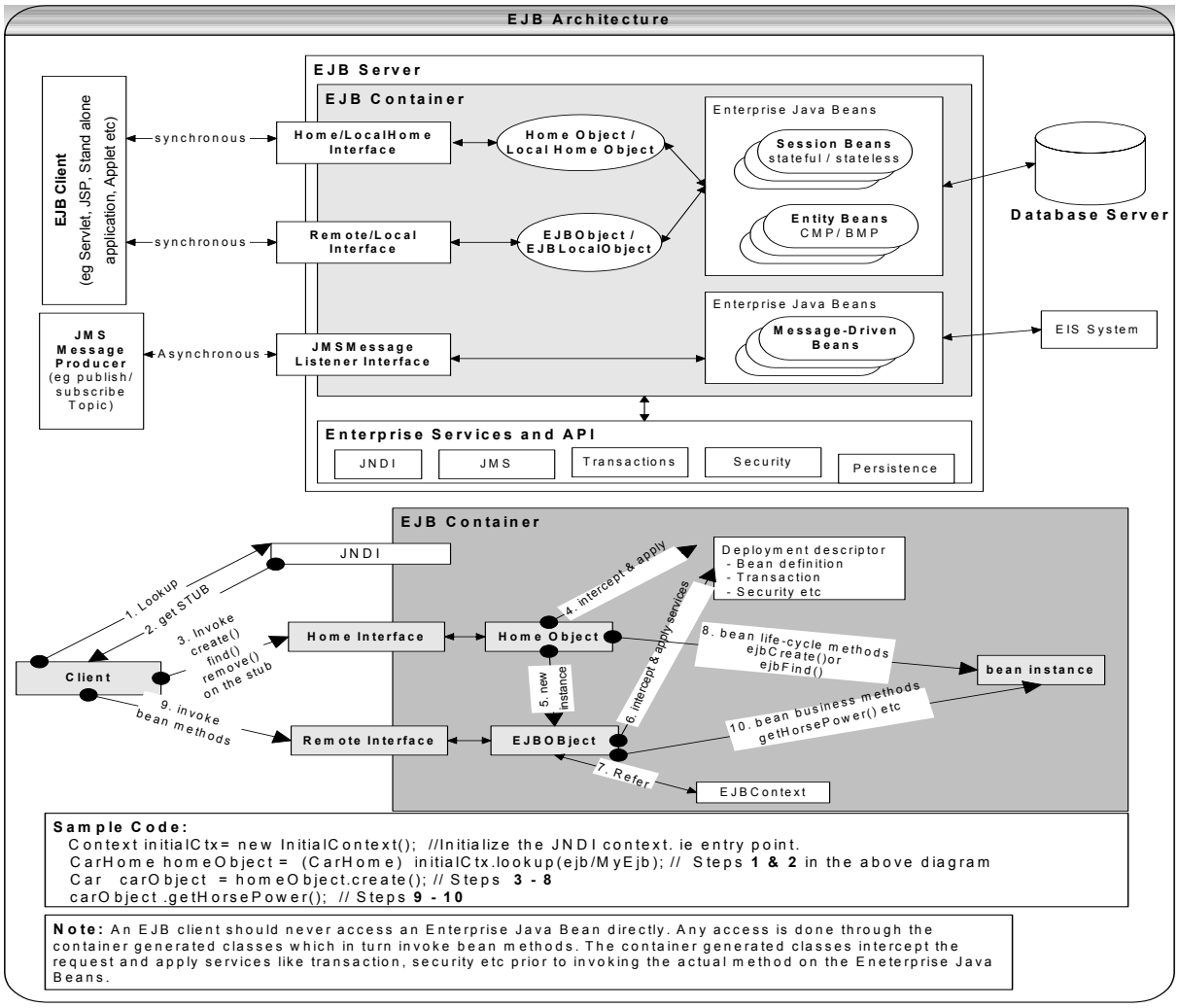
A 61: Both EJB and JavaBeans have very similar names but this is where the similarities end.

JavaBeans	Enterprise JavaBeans (EJB)
The components built based on JavaBeans live in a single local JVM (or address space) and can be either visual (e.g. GUI components like Button, List etc) or non-visual at runtime.	The Enterprise JavaBeans are non-visual distributable components, which can live across multiple JVMs (or address spaces).

No explicit support exists for services like transactions etc.	EJBs can be transactional and the EJB servers provide transactional support.
JavaBeans are fine-grained components, which can be used to assemble coarse-grained components or an application.	EJBs are coarse-grained components that can be deployed as is or assembled with other components into larger applications. EJBs must be deployed in a container that provides services like instance pooling, multi-threading, security, life-cycle management, transactions etc
Must conform to JavaBeans specification.	Must conform to EJB specification.

Q 62: Explain EJB architecture? **SF**

A 62:



EJB Container: EJBs are software components, which run in an environment called an EJB container. An EJB cannot function outside an EJB Container. The EJB container hosts and manages an Enterprise JavaBean in a similar manner that a Web container hosts a servlet or a Web browser hosts a Java Applet. The EJB container manages the following services so that the developer can concentrate on writing the business logic:

- Transactions (refer **Q71 – Q75** in Enterprise section)
- Persistence
- EJB instance pooling
- Security (refer **Q81** in Enterprise section)
- Concurrent access (or multi-threading)
- Remote access

Design pattern: EJBs use the proxy design pattern to make remote invocation (i.e. remote proxy) and to add container managed services like security and transaction demarcation. Refer **Q11** in "How would you about…" section for a more detailed discussion on proxy design pattern and dynamic proxies.

EJBContext: Every bean obtains an EJBContext object, which is a reference directly to the container. The EJB can request information about its environment like the status of a transaction, a remote reference to itself (an EJB cannot use 'this' to reference itself) etc.

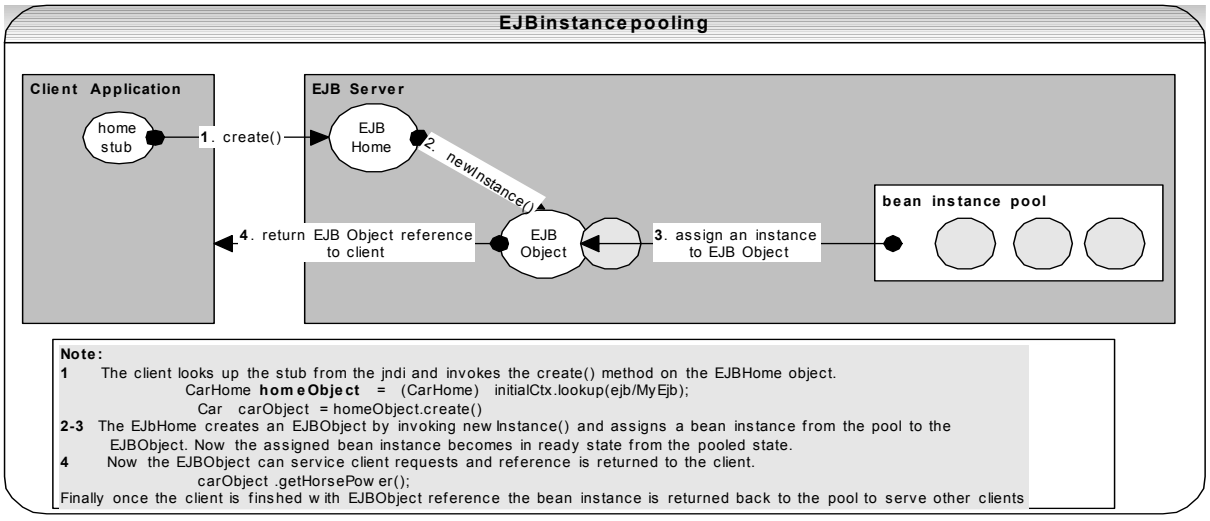
Deployment Descriptor: The container handles all the above mentioned services declaratively for an EJB based on the XML deployment descriptor (**ejb-jar.xml**). When an EJB is deployed into a container the deployment descriptor is read to find out how these services are handled. Refer to the *J2EE deployment structure* diagram in **Q6** in Enterprise section.

EJB: The EJB architecture defines 3 distinct types of Enterprise JavaBeans.

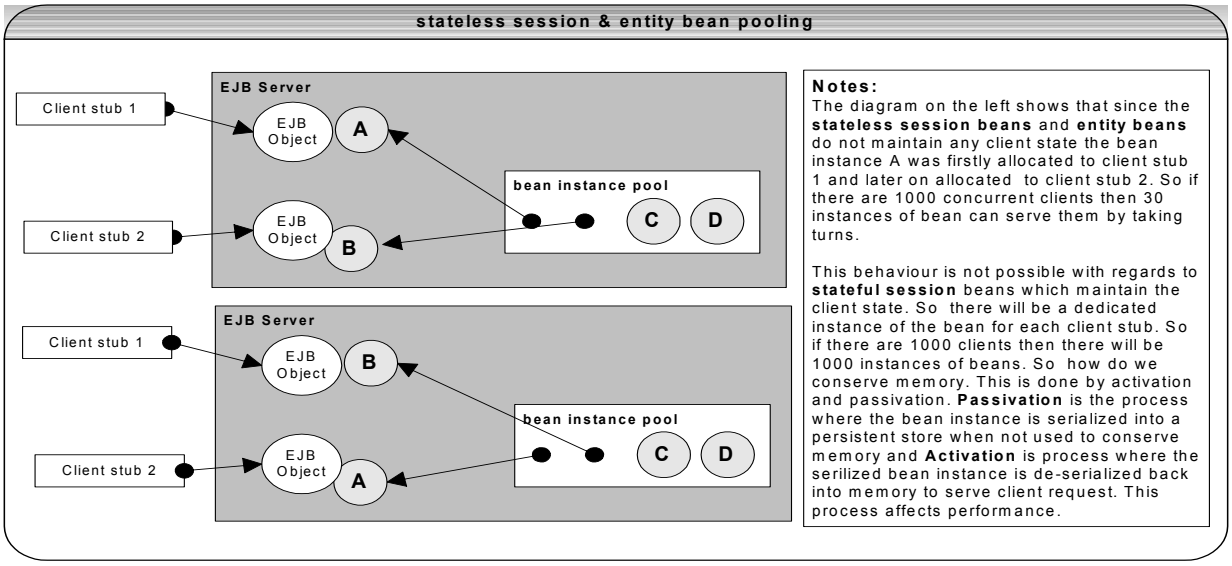
- Session beans.
- Entity beans.
- Message-driven beans.

The session and entity beans are invoked synchronously by the client and message driven beans are invoked asynchronously by a message container such as a publish/subscribe topic. Let's look at some of the EJB container services in a bit more detail:

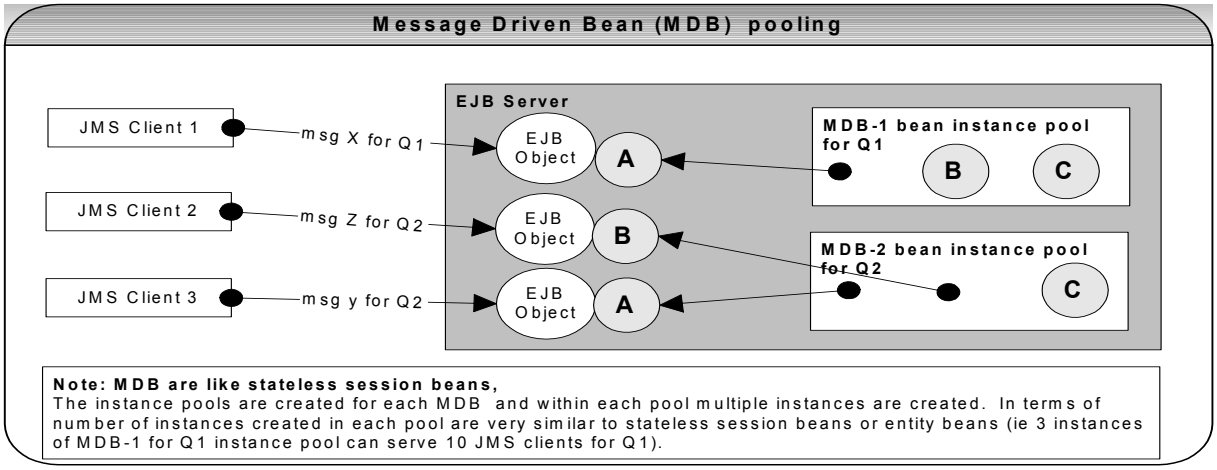
Instance pooling



The above diagram shows how the EJB instances are pooled and assigned to EJB Object and then returned to the pool. Let's look at in detail for different types of EJBs.



From the diagrams it is clear that bean instances can be reused for all the bean types except for the stateful session bean where the client state is maintained. So we need a dedicated stateful session bean for each client.



Concurrent access

The session beans do not support concurrent access. The stateful session beans are exclusively for a client so there is no concurrent access. The stateless session beans do not maintain any state. It does not make any sense to have concurrent access. The Entity beans represent data that is in the database table, which is shared between the clients. So to make concurrent access possible the EJB container need to protect the data while allowing many clients simultaneous access. When you try to share distributed objects you may have the following problem:

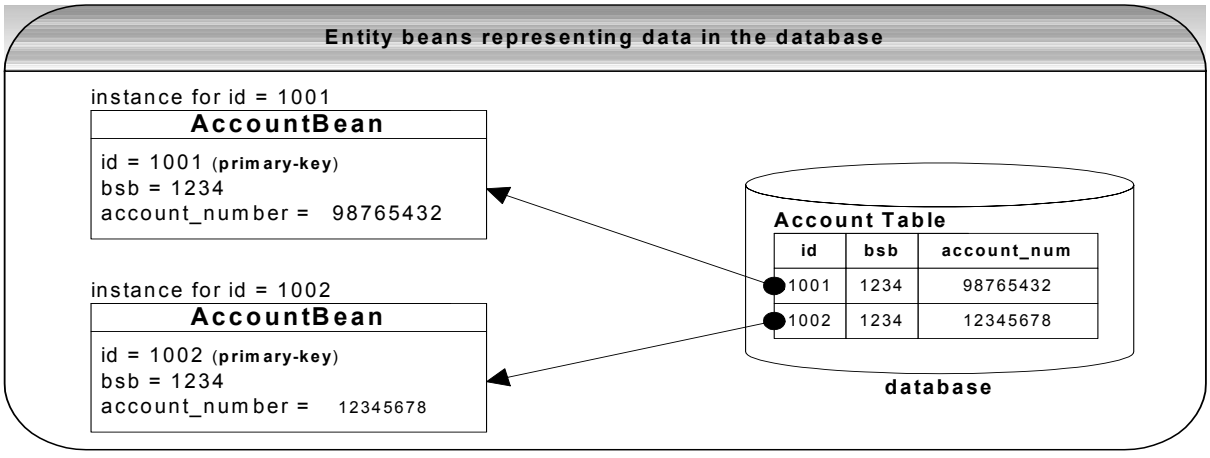
If 2 clients are using the same EJBObject, how do you keep one client from writing over the changes of the other? Say for example

```
Client-1 reads a value x= 5
Client-2 modifies the value to x=7
Now the client-1's value is invalid.
```

The entity bean addresses this by prohibiting concurrent access to bean instances. Which means several clients can be connected to one EJBObject but only one client can access the EJB instance at a time.

Persistence

Entity beans basically represent the data in a relational database. An Entity Bean is responsible for keeping its state in sync with the database.



- Container-managed persistence (CMP) - The container is responsible for saving the bean's state with the help of object-relational mapping tools.
- Bean-managed persistence (BMP) – The Entity Bean is responsible for saving its own state.

If entity beans performance is of concern then there are other persistence technologies and frameworks like JDBC, JDO, Hibernate, OJB and Oracle TopLink (commercial product).

Q 63: What are the different kinds of enterprise beans? SF

A 63:

Session Bean: is a non-persistent object that implements some business logic running on the server. Session beans do not survive system shut down. There are two types of session beans

- Stateless session beans (each session bean can be reused by multiple EJB clients)
- Stateful session beans (each session bean is associated with one EJB client)

Entity Bean: is a persistent object that represents object views of the data, usually a row in a database. They have the primary key as a unique identifier. Multiple EJB clients can share each entity bean. Entity beans can survive system shut shutdowns. Entity beans can have two types of persistence

- Container-managed persistence (CMP) - The container is responsible for saving the bean's state.
- Bean-managed persistence (BMP) – The Entity Bean is responsible for saving its own state.

Message-driven Bean: is integrated with the Java Message Service (JMS) to provide the ability to act as a message consumer and perform asynchronous processing between the server and the message producer.

Q 64: What is the difference between session and entity beans? SF

A 64:

Session Beans	Entity Beans
Use session beans for application logic.	Use entity beans to develop persistent object model.
Expect little reuse of session beans.	Insist on reuse of entity beans.
Session beans control the workflow and transactions of a group of entity beans.	Domain objects with a unique identity (ie-primary key) shared by multiple clients.
Life is limited to the life of a particular client. Handle database access for a particular client.	Persist across multiple invocations. Handles database access for multiple clients.
Do not survive system shut downs or server crashes.	Do survive system shut downs or server crashes.

Q 65: What is the difference between stateful and stateless session beans? SF

A 65:

Stateless Session Beans	Stateful Session Bean
Do not have an internal state. Can be reused by different clients.	Do have an internal state. Reused by the same client.
Need not be activated or passivated since the beans are pooled and reused.	Need to handle activation and passivation to conserve system memory since one session bean object per client.

Q 66: What is the difference between Container Managed Persistence (CMP) and Bean Managed Persistence (BMP)? SF

A 66:

Container Managed Persistence (CMP)	Bean Managed Persistence (BMP)
The container is responsible for persisting state of the bean.	The bean is itself responsible for persisting its own state.
Container needs to generate database (SQL) calls.	The bean needs to code its own database (SQL) calls.
The bean persistence is independent of its database (e.g. DB2, Oracle, Sybase etc). So it is portable from one data source to another.	The bean persistence is hard coded and hence may not be portable between different databases (e.g. DB2, Oracle etc).

Q 67: Can an EJB client invoke a method on a bean directly? SF

A 67: An EJB client should never access an EJB directly. Any access is done through the container. The container will intercept the client call and apply services like transaction, security etc prior to invoking the actual EJB. This relationship between the EJB and the container is like "**don't call us, we will call you**".

Q 68: How does an EJB interact with its container and what are the call-back methods in entity beans? SF

A 68: EJB interacts with its container through the following mechanisms

- **Call-back Methods**: Every EJB implements an interface (extends EnterpriseBean) which defines several methods which alert the bean to various events in its lifecycle. A container is responsible for invoking these methods. These methods notify the bean when it is about to be activated, to be persisted to the database, to end a transaction, to remove the bean from the memory, etc. For example the entity bean has the following call-back methods:

```
public interface javax.ejb.EntityBean {

   public void setEntityContext(javax.ejb.EntityContext c);
   public void unsetEntityContext();
   public void ejbLoad();
   public void ejbStore();
   public void ejbActivate();
   public void ejbPassivate();
   public void ejbRemove();
}
```

- **EJBContext:** provides methods for interacting with the container so that the bean can request information about its environment like the identity of the caller, security, status of a transaction, obtains remote reference to itself etc. e.g. isUserInRole(), getUserPrincipal(), isRollbackOnly(), etc

- **JNDI (Java Naming and Directory Interface):** allows EJB to access resources like JDBC connections, JMS topics and queues, other EJBs etc.

Q 69: What is the difference between EJB 1.1 and EJB 2.0? What is the difference between EJB 2.x and EJB 3.0? **SF**
A 69: EJB 2.0 has the following additional advantages over the EJB 1.1

- **Local interfaces**: These are beans that can be used locally, that means by the same Java Virtual Machines, so they do not required to be wrapped like remote beans, and arguments between those interfaces are passed directly by reference instead of by value. This improves performance.

- **ejbHome methods**: Entity beans can declare ejbHome methods that perform operations related to the EJB component but that are not specific to a bean instance.

- **Message Driven Beans (MDB)**: is a completely new enterprise bean type, which is designed specifically to handle incoming JMS messages.

- **New CMP Model**. It is based on a new contract called *the abstract persistence schema*, which will allow to the container to handle the persistence automatically at runtime.

- **EJB Query Language:** It is a sql-based language that will allow the new persistence schema to implement and execute finder methods.

Let's look at some of the new features on EJB 2.1

- **Container-managed timer service**: The timer service provides coarse-grained, transactional, time-based event notifications to enable enterprise beans to model and manage higher-level business processes.

- **Web service support**: EJB 2.1 adds the ability of stateless session beans to implement a Web service endpoint via a Web service endpoint interface.

- **EJB-QL**: Enhanced EJB-QL includes support for aggregate functions and ordering of results.

Current **EJB 2.x** model is complex for a variety of reasons:

- You need to create several component interfaces and implement several unnecessary call-back methods.

- EJB deployment descriptors are complex and error prone.

- EJB components are not truly object oriented, as they have restrictions for using inheritance and polymorphism.

- EJB modules cannot be tested outside an EJB container and debugging an EJB inside a container is very difficult.

Note: EJB 3.0 is taking ease of development very seriously and has adjusted its model to offer the POJO (Plain Old Java Object) persistence and the new **O/R mapping model based on Hibernate**. In EJB 3.0, **all kinds of enterprise beans are just POJOs**. EJB 3.0 **extensively uses Java annotations**, which replaces excessive XML based configuration files and eliminate the need for rigid component model used in EJB 1.x, 2.x. Annotations can be used to define the bean's business interface, O/R mapping information, resource references etc. Refer **Q18** in Emerging Technologies/Frameworks section.

Q 70: What are the implicit services provide by an EJB container? SF

A 70:

- **Lifecycle Management**: Individual enterprise beans do not need to explicitly manage process allocation, thread management, object activation, or object destruction. The EJB container automatically manages the object lifecycle on behalf of the enterprise bean.
- **State Management:** Individual enterprise beans do not need to explicitly save or restore conversational object state between method calls. The EJB container automatically manages object state on behalf of the enterprise bean.
- **Security:** Individual enterprise beans do not need to explicitly authenticate users or check authorisation levels. The EJB container automatically performs all security checking on behalf of the enterprise bean.
- **Transactions:** Individual enterprise beans do not need to explicitly specify transaction demarcation code to participate in distributed transactions. The EJB container can automatically manage the start, enrolment, commitment, and rollback of transactions on behalf of the enterprise bean.
- **Persistence:** Individual enterprise beans do not need to explicitly retrieve or store persistent object data from a database. The EJB container can automatically manage persistent data on behalf of the enterprise bean.

Q 71: What are transactional attributes? SF TI

A 71: EJB transactions are a set of mechanisms and concepts, which insures the integrity and consistency of the database when multiple clients try to read/update the database simultaneously.

Transaction attributes are defined at different levels like EJB (or class), a method within a class or segment of a code within a method. The attributes specified for a particular method take precedence over the attributes specified for a particular EJB (or class). Transaction attributes are specified *declaratively* through EJB deployment descriptors. Unless there is any compelling reason, the *declarative* approach is recommended over programmatic approach where all the transactions are handled programmatically. With the *declarative* approach, the EJB container will handle the transactions.

Transaction Attributes	Description
Required	Methods executed within a transaction. If client provides a transaction, it is used. If not, a new transaction is generated. Commit at end of method that started the transaction. Which means a method that has *Required* attribute set, but was called when the transaction has already started will not commit at the method completion. Well suited for EJB session beans.
Mandatory	Client of this EJB must create a transaction in which this method operates, otherwise an error will be reported. Well-suited for entity beans.
RequiresNew	Methods executed within a transaction. If client provides a transaction, it is suspended. If not a new transaction is generated, regardless. Commit at end of method.
Supports	Transactions are optional.
NotSupported	Transactions are not supported. If provided, ignored.
Never	Code in the EJB responsible for explicit transaction control.

Q 72: What are isolation levels? SF TI PI

A 72: Isolation levels provide a degree of control of the effects one transaction can have on another concurrent transaction. Since concurrent effects are determined by the precise ways in which, a particular relational database handles locks and its drivers may handle these locks differently. The semantics of isolation mechanisms based on these are not well defined. Nevertheless, certain defined or approximate properties can be specified as follows:

Isolation level	Description
TRANSACTION_SERIALIZABLE	Strongest level of isolation. Places a range lock on the data set, preventing other users from updating or inserting rows into the data set until the transaction is complete. Can produce deadlocks.

TRANSACTION_REPEATABLE_READ	Locks are placed on all data that is used in a query, preventing other users from updating the data, but new **phantom records** can be inserted into the data set by another user and are included in later reads in the current transaction.
TRANSACTION_READ_COMMITTED	Can't read uncommitted data by another transaction. Shared locks are held while the data is being read to avoid **dirty reads,** but the data can be changed before the end of the transaction resulting in **non-repeatable reads** and **phantom records.**
TRANSACTION_READ_UNCOMMITTED	Can read uncommitted data (**dirty read**) by another transaction, and **non-repeatable reads** and **phantom records** are possible. Least restrictive of all isolation levels. No shared locks are issued and no exclusive locks are honoured.

Isolation levels are not part of the EJB specification. They can only be set on the resource manager either explicitly on the *Connection* (for bean managed persistence) or via the application server specific configuration. The EJB specification indicates that isolation level is part of the **Resource Manager**.

As the transaction **isolation level increases**, likely **performance degradation** follows, as additional locks are required to protect data integrity. If the underlying data does not require such a high degree of integrity, the isolation level can be lowered to improve performance.

Q 73: What is a distributed transaction? What is a 2-phase commit? SF TI

A 73: A **Transaction** (Refer **Q43** in Enterprise section) is a series of actions performed as a single unit of work in which either all of the actions performed as a logical unit of work in which, either all of the actions are performed or none of the actions. A transaction is often described by ACID properties (Atomic, Consistent, Isolated and Durable). A **distributed transaction** is an ACID transaction between two or more independent transactional resources like two separate databases. For the transaction to commit successfully, all of the individual resources must commit successfully. If any of them are unsuccessful, the transaction must rollback in all of the resources. A **2-phase commit** is an approach for committing a distributed transaction in 2 phases.

Phase 1 is prepare: Each of the resources votes on whether it's ready to commit – usually by going ahead and persisting the new data but not yet deleting the old data.

Phase 2 is committing: If all the resources are ready, they all commit – after which old data is deleted and transaction can no longer roll back. 2-phase commit ensures that a distributed transaction can always be committed or always rolled back if one of the databases crashes. The **XA** specification defines how an application program uses a transaction manager to coordinate distributed transactions across multiple resource managers. Any resource manager that adheres to XA specification can participate in a transaction coordinated by an XA-compliant transaction manager.

Q 74: What is dooming a transaction? TI

A 74: A transaction can be doomed by the following method call CO

```
EJBContext.setRollbackOnly();
```

The above call will force transaction to rollback. The doomed transactions decrease scalability and if a transaction is doomed why perform compute intensive operations? So we can detect a doomed transaction as shown below: CO

```
public void  doComputeIntensiveOperation()  throws Exception {

   If ( ejbContext.getRollbackOnly() ) {
     return; //transaction is doomed so return (why unnecessarily perform compute intensive operation )
   }
   else {
      performComplexOperation();
   }
}
```

Q 75: How to design transactional conversations with session beans? SF TI

A 75: A stateful session bean is a resource which has an in memory state which can be rolled back in case of any failure. It can participate in transactions by implementing SessionSynchronization. CO

SessionSynchronization

```
public class MyBean implements SessionBean, SessionSynchronization{
        public int oldVal ;   public int val ;

        public void ejbCreate(int val) throws CreateException {
                this.val=val;
                this.oldVal=val;
        }

        public void afterBegin() { this.oldVal = this.val ;}
        public void   beforeCompletion(){};
        public void afterCompletion(boolean b) {  if (b == false)  this.val = this.oldVal ; }
       .....
}
```

```
public interface javax.ejb.SessionSynchronization {
    public void afterBegin();
    public void beforeCompletion();
    public void afterCompletion(boolean b);
}
```

The uses of SessionSynchronization are:

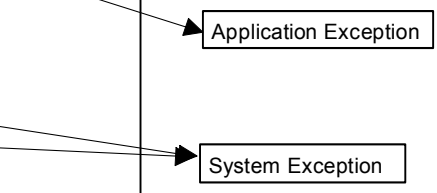
- Enables the bean to act as a transactional resource and undo state changes on failure.
- Enables you to cache database data to improve performance.

Q 76: Explain exception handling in EJB? SF EH CO
A 76: Java has two types of exceptions:

- **Checked exception**: derived from **java.lang.Exception** but not **java.lang.RuntimeException**.
- **Unchecked exception:** derived from **java.lang.RuntimeException** thrown by JVM.

System vs Application Exception

```
public void depositAmount()  throws InsufficientFundException {
        if(this.amount <= 0) {
           throw new InsufficientFundException ("Balance is <= 0");
        }
        try {
           depositAmount();
        } catch (SQLException e) {
             throw new EJBException(e);
        } catch (Exception e) {
             throw new EJBException(e);
        }
}
```



EJB has two types of exceptions:

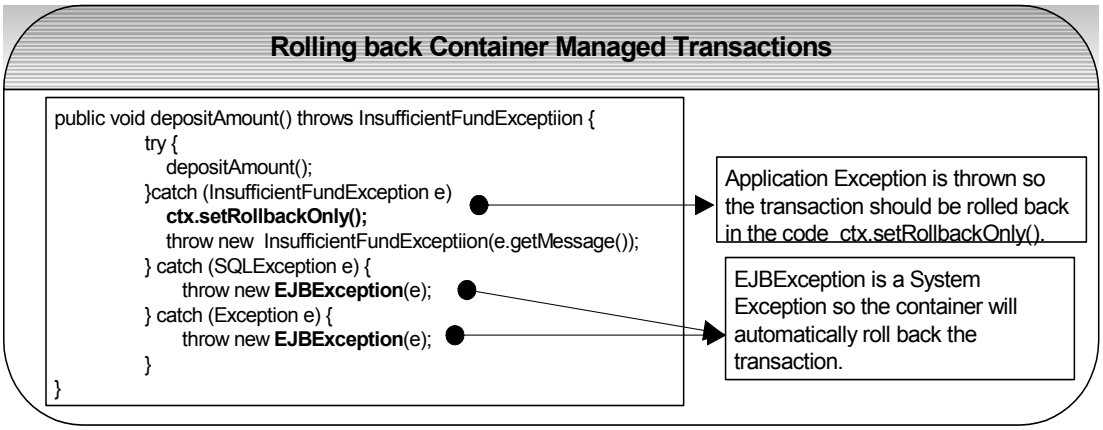
- **System Exception**: is an **unchecked** exception derived from **java.lang.RuntimeException**.
- **Application Exception**: is specific to an application and thrown because of violation of business rules.

A **System Exception** is thrown by the system and is not recoverable. For example EJB container losing connection to the database server, failed remote method objects call etc. Because the System Exceptions are unpredictable, the EJB container is the only one responsible for trapping the System Exceptions. The container automatically wraps any *RuntimeException* in **RemoteException**, which subsequently gets thrown to the caller (or client). In addition to intercepting System Exception the container may log the errors.

An **Application Exception** is specific to an application and is thrown because of violation of business rules. The client should be able to determine how to handle an Application Exception. If the account balance is zero then an Application Exception like **InsufficientFundException** can be thrown. If an **Application Exception** should be treated as a System Exception (e.g. SQLException) it needs to be wrapped in an **EJBException** so that it can be managed properly and propagated to the client.

Q 77: How do you rollback a container managed transaction in EJB? SF TI EH

A 77: The way the exceptions are handled affects the way the transactions are managed. CO



When the container manages the transaction, it is automatically rolled back when a **System Exception** occurs. This is possible because the container can intercept System Exception. However when an **Application Exception** occurs, the container does not intercept it and therefore leaves it to the code to roll back using **ctx.setRollbackOnly()**.

Be aware that handling exceptions in EJB is different from handling exceptions in Java. The Exception handling best practice tips are:

- If you cannot recover from System Exception let the container handle it.
- If a business rule is violated then throw an application exception.
- Catch the Exceptions in a proper order.
- It is a poor practice to catch **java.lang.Exception** because this is a big basket, which will catch all the unhandled exceptions. It is shown in the above diagrams for illustration purpose only. You should avoid this because if you add a new piece of code, which throws a new, checked exception, then the compiler won't pick it up.

Q 78: What is the difference between optimistic and pessimistic concurrency control? TI

A 78:

Pessimistic Concurrency	Optimistic Concurrency
A pessimistic design assumes conflicts will occur in the database tables and avoids them through exclusive locks etc.	An optimistic approach assumes conflicts won't occur, and deal with them when they do occur.
EJB (also non-EJB) locks the source data until it completes its transaction. ▪ Provides reliable access to data. ▪ Suitable for short transactions. ▪ Suitable for systems where concurrent access is rare.	EJB (also non-EJB) implements a strategy to detect whether a change has occurred. Locks are placed on the database only for a small portion of the time. ▪ Suitable for long transactions. ▪ Suitable for systems requiring frequent concurrent accesses.
The pessimistic locking imposes high locking overheads on the server and lower concurrency.	The optimistic locking is used in the context of cursors. The optimistic locking works as follows: ▪ No locks are acquired as rows are read. ▪ No locks are acquired while values in the current row are changed. ▪ When changes are saved, a copy of the row in the database is read in the locked mode. ▪ If the data was changed after it was read into the cursor, an error is raised so that the transaction can be rolled back and retried. **Note: The testing for changes can be done by comparing the values, timestamp or version numbers**.

Q 79: How can we determine if the data is stale (for example when using optimistic locking)? TI

A 79: We can use the following strategy to determine if the data is stale:

- Adding version numbers

1. Add a version number (Integer) to the underlying table.
2. Carry the version number along with any data read into memory (through value object, entity bean etc).
3. Before performing any update compare the current version number with the database version number.
4. If the version numbers are equal update the data and increment the version number.
5. If the value object or entity bean is carrying an older version number, reject the update and throw an exception.

Note: You can also do the version number check as part of the update by including the version column in the where clause of the update without doing a prior select.

- Adding a timestamp to the underlying database table.
- Comparing the data values.

These techniques are also quite useful when implementing data caching to improve performance. Data caches should regularly keep track of stale data to refresh the cache. These strategies are valid whether you use EJB or other persistence mechanisms like JDBC, Hibernate etc.

Q 80: What are not allowed within the EJB container? **SF**

A 80: In order to develop reliable and portable EJB components, the following restrictions apply to EJB code implementation:

- Avoid using static non-final fields. Declaring all static fields in EJB component as final is recommended. This enables the EJB container to distribute instances across multiple JVMs.
- Avoid starting a new thread (conflicts with EJB container) or using thread synchronization (allow the EJB container to distribute instances across multiple JVMs).
- Avoid using AWT or Swing functionality. EJBs are server side business components.
- Avoid using file access/java.io operations. EJB business components are meant to use resource managers such as JDBC to store and retrieve application data. Also deployment descriptors can be used to store <env-entry>.
- Avoid accepting or listening to socket connections. EJB components are not meant to provide network socket functionality. However the specification lets EJB components act as socket clients or RMI clients.
- Avoid using the reflection API. This restriction enforces Java security.
- Can't use custom class loaders.

Q 81: Discuss EJB container security? **SF** **SE**

A 81: EJB components operate inside a container environment and rely heavily on the container to provide security. The four key services required for the security are:

- **Identification**: In Java security APIs this identifier is known as a **principal**.
- **Authentication**: To prove the identity one must present the credentials in the form of password, swipe card, digital certificate, finger prints etc.
- **Authorisation (Access Control)**: Every secure system should limit access to particular users. The common way to enforce access control is by maintaining **security roles** and **privileges**.
- **Data Confidentiality**: This is maintained by encryption of some sort. It is no good to protect your data by authentication if someone can read the password.

The EJB specification concerns itself exclusively with **authorisation** (access control). An application using EJB can specify in an abstract (declarative) and portable way that is allowed to access business methods. The EJB container handles the following actions:

- Find out the Identity of the caller of a business method.

- Check the EJB deployment descriptor to see if the identity is a member of a security role that has been granted the right to call this business method.
- Throw java.rmi.RemoteException if the access is illegal.
- Make the identity and the security role information available for a fine grained programmatic security check.

```
public  void closeAccount() {
    if (ejbContext.getCallerPrincipal().getName() = “SMITH”) {
      //…
    }

    if (!ejbContext .isCallerInRole(CORPORATE_ACCOUNT_MANAGER)) {
      throw new SecurityException(“Not authorized to close this account”);
    }
}
```

- Optionally log any illegal access.

There are two types of information the EJB developer has to provide through the deployment descriptor.

- Security roles
- Method permissions

Example:

```
<security-role>
  <description>
    Allowed to open and close accounts
  </description>
  <role-name>account_manager</role-name>
</security-role>
<security-role>
  <description>
    Allowed to read only
  </description>
  <role-name>teller</role-name>
</security-role>
```

There is a many-to-many relationship between the security roles and the method permissions.

```
<method-permission>
  <role-name>teller</role-name>
  <method>
    <ejb-name>AccountProcessor</ejb-name>
    <method-name>findByPrimaryKey</method-name>
 </method>
</method-permission>
```

Just as we must declare the resources accessed in our code for other EJBs that we reference in our code we should also declare the security role we access programmatically to have a fine grained control as shown below.

```
<security-role-ref>
  <description>
    Allowed to open and close accounts
  </description>
 <role-name>account_manager</role-name>
 <role-link>executive</role-link>
</security-role-ref>
```

There is also many-to-many relationship between the EJB specific security roles that are in the deployment descriptor and the application based target security system like LDAP etc. For example there might be more than one group users and individual users that need to be mapped to a particular EJB security role ‘account_manager’.

Q 82: What are EJB best practices? **BP**

A 82:

- Use local interfaces that are available in EJB2.0 if you deploy both the EJB client and the EJB in the same server. Use vendor specific pass-by-reference implementation to make EJB1.1 remote EJBs operate as local.

[Extreme care should be taken not to affect the functionality by switching the application, which was written and tested in pass-by-reference mode to pass-by-value without analysing the implications and re-testing the functionality.

- Wrap entity beans with session beans to reduce network calls (refer **Q84** in Enterprise section) and promote declarative transactions. Where possible use local entity beans and session beans can be either local or remote. Apply the appropriate EJB design patterns as described in **Q83 – Q87** in Enterprise section.
- Cache ejbHome references to avoid JNDI look-up overhead using service locator pattern.
- Handle exceptions appropriately (refer **Q76, Q77** in Enterprise section).
- Avoid transaction overhead for non-transactional methods of session beans by declaring transactional attribute as 'Supports'.
- Choose plain Java object over EJB if you do not want services like RMI/IIOP, transactions, security, persistence, thread safety etc. There are alternative frameworks such as Hibernate, Spring etc.
- Choose Servlet's **HttpSession** object rather than stateful session bean to maintain client state if you do not require component architecture of a stateful bean.
- Apply **Lazy loading** and **Dirty marker** strategies as described in **Q88** in Enterprise section.

Session Bean (stateless)	Session Bean (stateful)	Entity Bean
▪ Tune the pool size to avoid overhead of creation and destruction. ▪ Use setSessionContext(..) or ejbCreate(..) method to cache any bean specific resources. ▪ Release any acquired resources like Database connection etc in ejbRemove() method	▪ Tune the pool size to avoid overhead of creation and destruction. ▪ Set proper time out to avoid resource congestion. ▪ Remove it explicitly from client using remove() method. ▪ Use 'transient' variable where possible to avoid serialization overhead.	▪ Tune the pool size to avoid overhead of creation and destruction. ▪ Use setEntityContext(..) method to cache any bean specific resources and unsetEntityContext() method to release acquired resources. ▪ Use lazy-loading to avoid any unnecessary loading of dependent data. Use dirty marker to avoid unchanged data update. ▪ Commit the data after a transaction completes to reduce any database calls in between. ▪ Where possible perform bulk updates, use CMP rather than BMP, Use direct JDBC (Fast-lane-reader) instead of entity beans, use of read-only entity beans etc.

Q 83: What is a business delegate? Why should you use a business delegate? **DP** **PI**

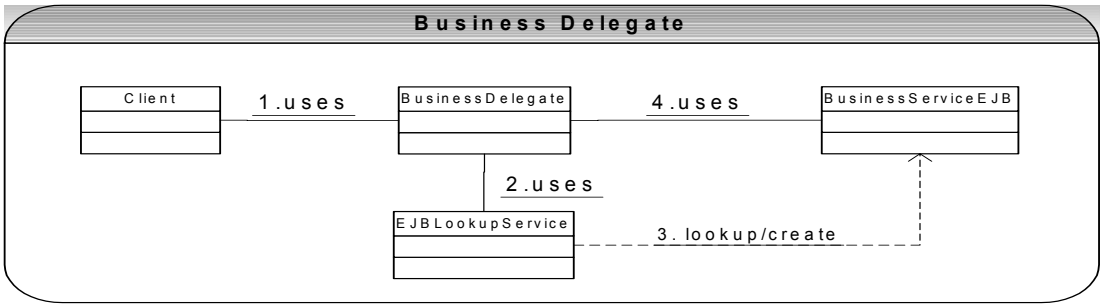
A 83: Questions **Q83 – Q88** are very popular EJB questions.

Problem: When presentation tier components interact directly with the business services components like EJB, the presentation components are vulnerable to changes in the implementation of business services components.

Solution: Use a **Business Delegate** to reduce the coupling between the presentation tier components and the business services tier components. Business Delegate hides the underlying implementation details of the business service, such as look-up and access details of the EJB architecture.

Business delegate **is responsible for:**

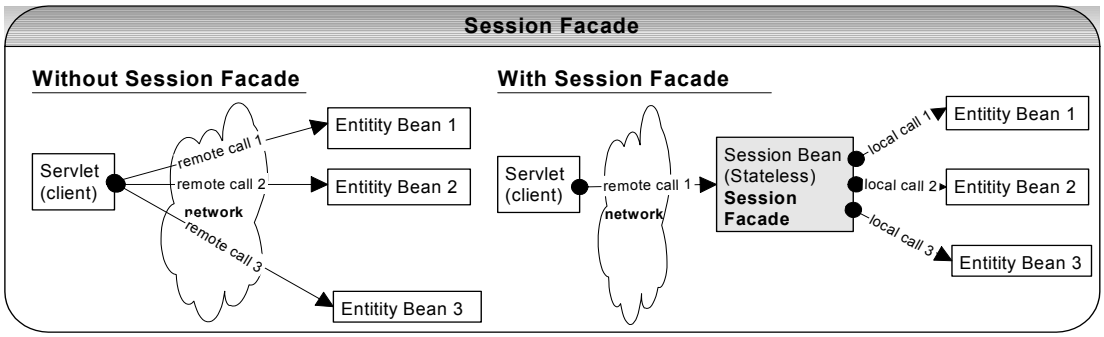
- Invoking session beans in Session Facade.
- Acting as a service locator and cache home stubs to improve performance.
- Handling exceptions from the server side. (Unchecked exceptions get wrapped into the remote exception, checked exceptions can be thrown as an application exception or wrapped in the remote exception. unchecked exceptions do not have to be caught but can be caught and should not be used in the method signature.)
- Re-trying services for the client (For example when using optimistic locking business delegate will retry the method call when there is a concurrent access.).



Q 84: What is a session façade? DP PI

A 84: **Problem**: Too many method invocations between the client and the server will lead to network overhead, tight coupling due to dependencies between the client and the server, misuse of server business methods due to fine grained access etc.

Solution: Use a **session** bean as a **façade** to encapsulate the complexities between the client and the server interactions. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.



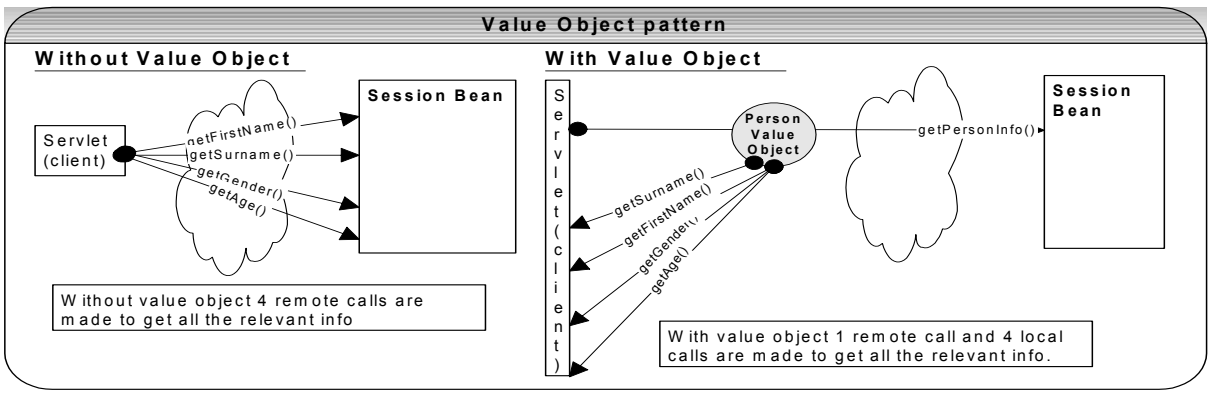
Session façade is responsible for

- Improving performance by minimising fine-grained method calls over the network.
- Improving manageability by reducing coupling, exposing uniform interface and exposing fewer methods to clients.
- Managing transaction and security in a centralised manner.

Q 85: What is a value object pattern? DP PI

A 85: **Problem**: When a client makes a remote call to the server, there will be a process of network call and serialization of data involved for the remote invocation. If you make fine grained calls there will be performance degradation.

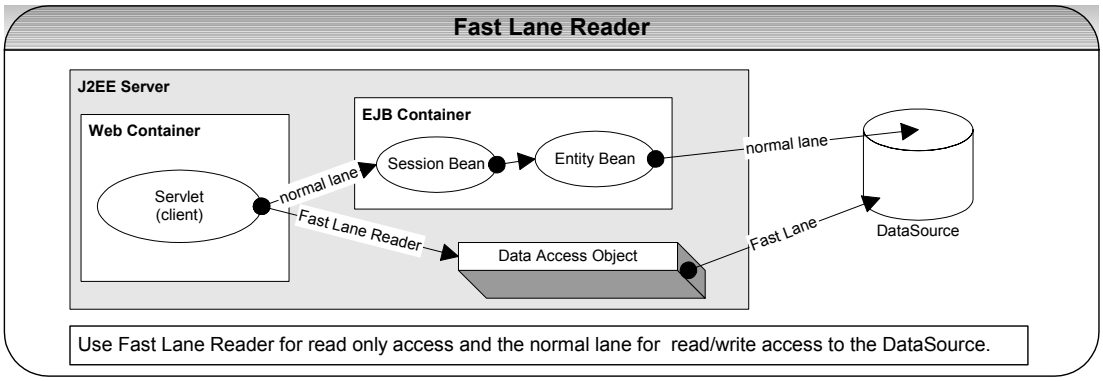
Solution: Avoid fine-grained method calls by creating a value object, which will help the client, make a coarse-grained call.



Q 86: What is a fast-lane reader? DP PI

A 86: **Problem**: Using Entity beans to represent persistent, read only tabular data incurs performance cost at no benefit (especially when large amount of data to be read).

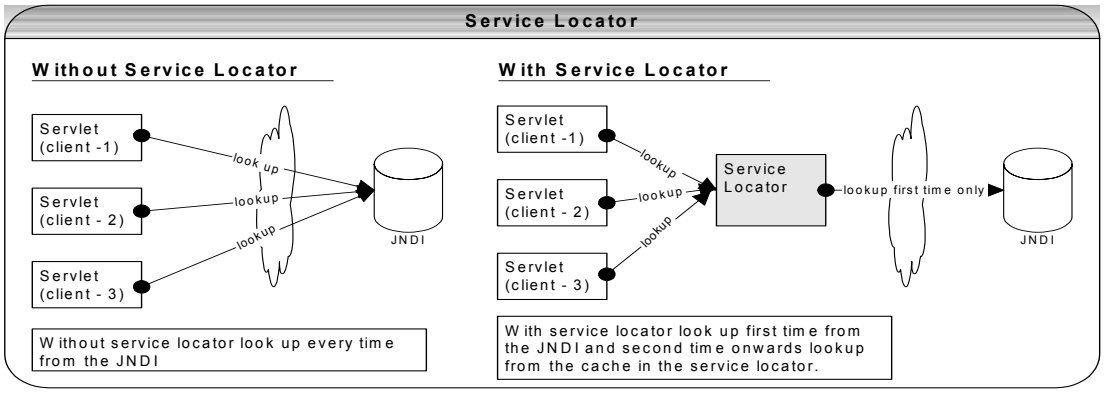
Solution: Access the persistent data directly from the database using the DAO (Data Access Object) pattern instead of using Entity beans. The Fast lane readers commonly use JDBC, Connectors etc to access the read-only data from the data source. The main benefit of this pattern is the faster data retrieval.



Q 87: What is a Service Locator? DP PI

A 87: **Problem**: J2EE makes use of the JNDI interface to access different resources like JDBC, JMS, EJB etc. The client looks up for these resources through the JNDI look-up. The JNDI look-up is expensive because the client needs to get a network connection to the server first. So this look-up process is expensive and redundant.

Solution: To avoid this expensive and redundant process, service objects can be cached when a client performs the JNDI look-up for the first time and reuse that service object from the cache for the subsequent look-ups. The service locator pattern implements this technique. Refer to diagram below:



Q 88: Explain lazy loading and dirty marker strategies? DP PI

A 88: **Lazy Loading**: Lazy loading means not creating an object until the first time it is accessed. This technique is useful when you have large hierarchies of objects. You can lazy load some of the dependent objects. You only create the dependent (subordinate) objects only when you need them.

```
If ( this.data = null) {
   //lazy load data
}
```

For a CMP bean the default scenario is set to no lazy loading and the finder method will execute a single SQL select statement against the database. So, for example, with the findAllCustomers() method will retrieve all customer objects with all the CMP fields in each customer object.

If you turn on lazy loading then only the primary keys of the objects within the finder are returned. Only when you access the object, the container uploads the actual object based on the primary key. You may want to turn on the lazy loading feature if the number of objects that you are retrieving is so large that loading them all into local cache would adversely affect the performance. (**Note:** The implementation of lazy loading strategy may vary from container vendor to vendor).

Dirty Marker (Store optimisation): This strategy allows us to persist only the entity beans that have been modified. The dependent objects need not be persisted if they have not been modified. This is achieved by using a dirty flag to mark an object whose contents have been modified. The container will check every dependent object and will persist only those objects that are dirty. Once it is persisted its dirty flag will be cleared. (**Note:** The implementation of dirty marker strategy may vary from container vendor to vendor).

Note: If your job requires a very good understanding of EJB 2.x then following books are recommended:
- **Mastering Enterprise JavaBeans** – by Ed Roman
- **EJB Design Patterns** – by Floyd Marinescu

Enterprise - JMS

Q 89: What is Message Oriented Middleware? What is JMS? SF

A 89: Message Oriented Middleware (MOM) is generally defined as a software infrastructure that asynchronously communicates with other disparate systems through the production and consumption of messages. A message may be a request, a report, or an event sent from one part of an enterprise application to another.

Messaging enables loosely coupled distributed communication. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate and also they are not aware of each other. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only what message format and what destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which requires an application to know a remote application's methods.

Message Oriented Middleware (MOM) systems like MQSeries, MQSonic, etc are proprietary systems. Java Message Service (JMS) is a Java API that allows applications to create, send, receive, and read messages in a standard way. Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations (e.g. MQSonic, TIBCO etc). The JMS API minimises the set of concepts a programmer must learn to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximise the portability of JMS applications across JMS providers.

Companies have spent decades developing their legacy systems. Rather than throwing these systems out, XML can be used in a non-proprietary way to move data from legacy systems to distributed systems like J2EE over the wire-using MOM and JMS.

How JMS is different from RPC?

Remote Procedure Call (e.g. RMI)	JMS
Remote Procedure Call (RPC) technologies like RMI attempt to mimic the behaviour of system that runs in one process. When a remote procedure is invoked the caller is blocked until the procedure completes and returns control to the caller. This is a synchronized model where process is performed sequentially ensuring that tasks are completed in a predefined order. The synchronized nature of RPC tightly couples the client (the software making the call) to the server (the software servicing the call). The client can not proceed (its blocked) until the server responds. The tightly coupled nature of RPC creates highly interdependent systems where a failure on one system has an immediate impact on other systems.	With the use of Message Oriented Middleware (MOM), problems with the availability of subsystems are less of an issue. A fundamental concept of MOM is that communications between components is intended to be asynchronous in nature. Code that is written to connect the pieces together assumes that there is a one-way message that requires no immediate response. In other words, there is no blocking. Once a message is sent the sender can move on to other tasks; it doesn't have to wait for a response. This is the major difference between RPC and asynchronous messaging and is critical to understanding the advantages offered by MOM systems. In an asynchronous messaging system each subsystem (Customer, Account etc) is decoupled from the other systems. They communicate through the messaging server, so that a failure in one does not impact the operation of the others.

Client is blocked while it is being processed.	Asynchronous messages also allows for parallel processing i.e. client can continue processing while the previous request is being satisfied.

Are messaging applications slow? While there is some overhead in all messaging systems, but this does not mean that the applications that are using messaging are necessarily slow. Messaging systems can achieve a throughput of 100 messages per second depending on the installation, messaging modes (synchronous versus asynchronous, persistent versus non-persistent), and acknowledgment options such as *auto mode*, *duplicates okay mode*, and *client mode* etc. The asynchronous mode can significantly boost performance by multi-tasking. **For example:** In an Internet based shopping cart application, while a customer is adding items to his/her shopping cart, your application can trigger an inventory checking component, and a customer data retrieval component to execute concurrently.

Are messaging applications reliable? This is basically a trade-off between performance and reliability. If reliability is more important then the:

- Acknowledgment option should be set to *automode* where once only delivery is guaranteed
- Message delivery mode should be set to *persistent* where the MOM writes the messages to a secure storage like a database or a file system to insure that the message is not lost in transit due to a system failure.

What are some of the key message characteristics defined in a message header?

Characteristic	Explanation
JMSCorrelationID	Used in request/response situations where a JMS client can use the JMSCorrelationID header to associate one message with another. **For example**: a client request can be matched with a response from a server based on the JMSCorrelationID.
JMSMessageID	Uniquely identifies a message in the MOM environment.
JMSDeliveryMode	This header field contains the delivery modes: PERSISTENT or NON_PERSISTENT.
JMSExpiration	This contains the time-to-live value for a message. If it is set to zero, then a message will never expire.
JMSPriority	Sets the message priority but the actual meaning of prioritization is MOM vendor dependent.

What are the different body types (aka **payload types) supported for messages?** All JMS messages are read-only once posted to a queue or a topic.

- **Text message**: body consists of java.lang.String.
- **Map message**: body consists of key-value pairs.
- **Stream message**: body consists of streams of Java primitive values, which are accessed sequentially. XML documents make use of this type.
- **Object message**: body consists of a Serializable Java object.
- **Byte message**: body consists of arbitrary stream of bytes.

What is a message broker?

A message broker acts as a server in a MOM. A message broker performs the following operations on a message it receives:

- Processes message header information.
- Performs security checks and encryption/decryption of a received message.
- Handles errors and exceptions.
- Routes message header and the payload (aka message body).
- Invokes a method with the payload contained in the incoming message (e.g. calling onMessage(..) method on a Message Driven Bean (MDB)).
- Transforms the message to some other format. For example XML payload can be converted to other formats like HTML etc with XSLT.

Q 90: What type of messaging is provided by JMS? **SF**

A 90: **Point-to-Point**: provides a traditional **queue** based mechanism where the client application sends a message through a queue to typically one receiving client that receives messages sequentially. A JMS message queue is an administered object that represents the message destination for the sender and the message source for the receiver.

Publish/Subscribe: is a one-to-many publishing model where client applications publish messages to **topics**, which are in turn subscribed by other interested clients. All subscribed clients will receive each message.

Q 91: Discuss some of the design decisions you need to make regarding your message delivery? SF DC

A 91:

During your design phase, you should carefully consider various options or modes like message acknowledgment modes, transaction modes and delivery modes. **For example**: for a simple approach you would not be using transactions and instead you would be using acknowledgment modes. If you need reliability then the delivery mode should be set to persistent. This can adversely affect performance but reliability is increased.

Design decision	Explanation
Message acknowledgment options or modes.	**Acknowledgement mode and transaction modes are used to determine if a message will be lost or re-delivered on failure during message processing by the target application**. Acknowledgment modes are set when creating a JMS session. InitialContext ic = new InitialContext(…); QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("AccountConnectionFactory"); QueueConnection qc = qcf.createQueueConnection(); **QueueSession session = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);** the above code sample, the transaction mode is set to false and acknowledgment mode is set to auto mode. Let us look at acknowledgment modes: **AUTO_ACKNOWLEDGE**: The messages sent or received from the session are automatically acknowledged. This mode also **guarantees once only delivery**. If a failure occurs while executing onMessage() method of the destination MDB, then the message is re-delivered. A message is automatically acknowledged when it successfully returns from the onMessage(…) method. **DUPS_OK_ACKNOWLEDGE**: This is just like AUTO_ACKNOWLEDGE mode, but under rare circumstances like during failure recovery messages might be delivered more than once. If a failure occurs then the message is re-delivered. This mode has fewer overheads than AUTO_ACKNOWLEDGE mode. **CLIENT_ACKNOWLEDGE**: The messages sent or received from sessions are not automatically acknowledged. The destination application must acknowledge the message receipt. This mode gives an application full control over message acknowledgment at the cost of increased complexity. This can be acknowledged by invoking the acknowledge() method on *javax.jms.Message* class.

Transaction modes	In JMS, a transaction organizes a message or a group of messages into an atomic processing unit. So, if a message delivery is failed, then the failed message may be re-delivered. Calling the commit() method commits all the messages the session receives and calling the rollback method rejects all the messages.
	
	InitialContext ic = new InitialContext(…);
	QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("AccountConnectionFactory");
	QueueConnection qc = qcf.createQueueConnection();
	QueueSession session = qc.createQueueSession(true, -1);
	
	In the above code sample, the transaction mode is set to true and acknowledgment mode is set to -1, which means acknowledgment mode has no use in this mode. Let us look at transaction modes:
	
	Message Driven Bean (MDB) with container managed transaction demarcation: An MDB participates in a container transaction by specifying the transaction attributes in its deployment descriptor. A transaction automatically starts when the JMS provider removes the message from the destination and delivers it to the MDB's onMessage(…) method. Transaction is committed on successful completion of the onMessage() method. A MDB can notify the container that a transaction should be rolled back by setting the *MessageDrivenContext* to **setRollBackOnly**(). When a transaction is rolled back, the message is re-delivered.
	
	public void **onMessage**(Message aMessage) {
	…
	If(someCondtionIsTrue) {
	mdbContext.setRollbackOnly();
	}
	else{
	//everything is good. Transaction will be **committed automatically** on completion of onMessage(..) method.
	}
	}
	
	Message Driven Bean (MDB) with bean managed transaction demarcation: A MDB chooses not to

<table>
<tr><td></td><td>
participate in a container managed transaction and the MDB programmer has to design and code programmatic transactions. This is achieved by creating a UserTransaction object from the MDB's MessageDrivenContext as shown below and then invoking the commit() and rollback() methods on this UserTransaction object.

<pre>
public void onMessage(Message aMessage) {

   UserTransaction uT = mdbContext.getUserTransaction();
   uT.begin();
   ….
   If(someCondtionIsTrue) {
     uT.rollback();
  }
  else{
    uT.commit();
  }
}
</pre>
Transacted session: An application completely controls the message delivery by either committing or rolling back the session. An application indicates successful message processing by invoking Session class's commit() method. Also it can reject a message by invoking Session class's rollback() method. This committing or rollback is applicable to all the messages received by the session.

<pre>
public void process(Message aMessage, QueueSession qs) {
   ….
  If(someCondtionIsTrue) {
     qs.rollback();
   }
  else{
     qs.commit();
  }
…
}
</pre>
What happens to rolled-back messages?

Rolled back messages are re-delivered based on the re-delivery count parameter set on the JMS provider. The re-delivery count parameter is very important because some messages can never be successful and this can eventually crash the system. When a message reaches its re-delivery count, the JMS provider can either log the message or forward the message to an error destination. Usually it is not advisable to retry delivering the message soon after it has been rolled-back because the target application might still not be ready. So we can specify a time to re-deliver parameter to delay the re-delivery process by certain amount of time. This time delay allows the JMS provider and the target application to recover to a stable operational condition.

Care should be taken not to make use of a single transaction when using the JMS request/response paradigm where a JMS message is sent, followed by the synchronous receipt of a reply to that message. This is because a JMS message is not delivered to its destination until the transaction commits, and the receipt of the reply will never take place within the same transaction.

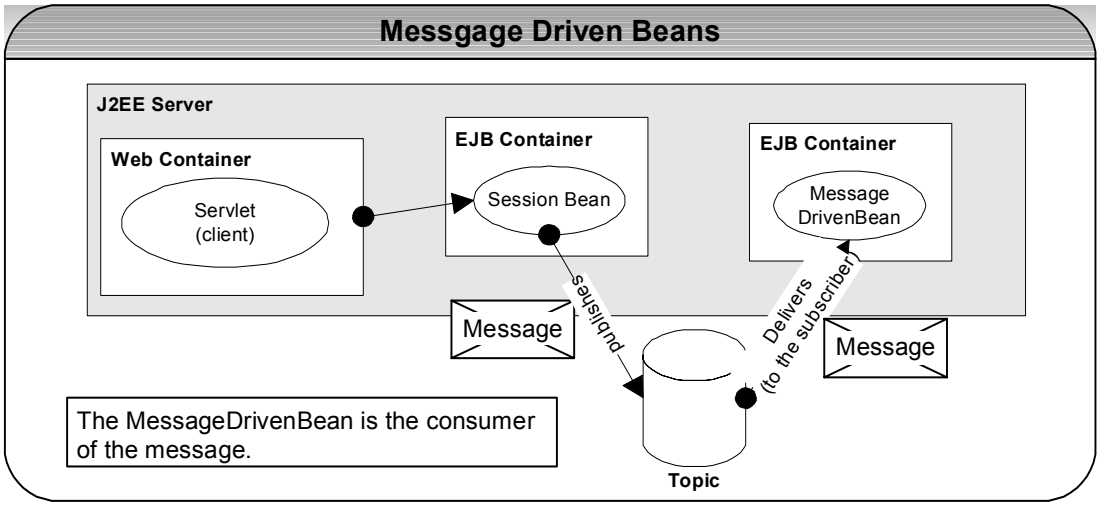
Note: when you perform a JNDI lookup for administered objects like connection factories, topics and/or queues, you should use the logical reference java:comp/env/jms as the environment subcontext. It is also vital to release the JMS resources like connection factories, sessions, queues, topics etc when they are no longer required in a try{} and finally{} block.
</td></tr>
<tr><td>Message delivery options</td><td>
What happens, when the messages are with the JMS provider (i.e. MOM) and a catastrophic failure occurs prior to delivering the messages to the destination application? The messages will be lost if they are non-durable. The message's state whether they are lost or not does not depend on acknowledgment modes or transaction modes discussed above. It depends on the delivery mode, which defines whether the message can be durable (aka persistent) or non-durable (aka non-persistent). If you choose the durable delivery mode then the message is stored into a database or a file system by the JMS server before delivering it to the consumer. Durable messages have an adverse effect on performance, but ensure that message delivery is guaranteed.

<pre>
InitialContext ic = new InitialContext(…);
QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("AccountConnectionFactory");
QueueConnection qc = qcf.createQueueConnection();
QueueSession session = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
//senderQueue is an object of type javax.jms.Queue
QueueSender sender = session.createSender(senderQueue);
Sender.send(message, intDeliveryMode, intPriority, longTimeToLove);
</pre>
</td></tr>
<tr><td>Best practices</td><td>
▪ A JMS connection represents a TCP/IP connection from the client to the JMS server. A connection is
</td></tr>
</table>

to improve performance	a valuable resource, which should be opened at the appropriate time, should be used concurrently by creating and using pool of sessions, and close the connection in a finally{} block when finished. ▪ Optimize your JMS sessions with the appropriate acknowledgment mode and transaction mode as discussed above and close your sessions when you are finished with them. ▪ Choose your message type (text message, byte message, stream message etc) carefully because the size of a message depends on its type and size can affect performance. For example byte messages takes less space than text messages and for object messages you can reduce the serialization cost by marking some of the variables which need not be sent over the network as transient. ▪ Optimize your destinations like queues and topics as follows: ▪ Choose a non-durable (aka non-persistent) delivery mode where appropriate. ▪ Set time to live parameter appropriately after which the message expires. ▪ Where applicable receive messages asynchronously (non-blocking call). If you want to receive messages synchronously you can use one of the following methods on the message consumer: receive(); → **blocks the call until it receives the next message.** receive(long timeout); → **blocks till a timeout occurs.** receiveNoWait(); → **never blocks**.

Q 92: Give an example of a J2EE application using Message Driven Bean with JMS? **SF**

A 92:



Enterprise - XML

What is XML? And why is XML important? XML stands for e**X**tensible **M**arkup **L**anguage. XML is a grammatical system for constructing custom markup languages for describing business data, mathematical data, chemical data etc. **XML loosely couples disparate applications or systems utilizing JMS, Web services** etc. XML uses the same building blocks that HTML does: elements, attributes and values. Let's look at why XML is important.

Scalable: Since XML is not in a binary format you can create and edit files with anything and it's also easy to debug. XML can be used to efficiently store small amounts of data like configuration files (web.xml, application.xml, struts-config.xml etc) to large company wide data with the help of XML stored in the database.

Fast Access: XML documents benefit from their hierarchical structure. Hierarchical structures are generally faster to access because you can drill down to the section you are interested in.

Easy to identify and use: XML not only displays the data but also tells you what kind of data you have. The mark up tags identifies and groups the information so that different information can be identified by different application.

Stylability: XML is style-free and whenever different styles of output are required the same XML can be used with different style-sheets (XSL) to produce output in XHTML, PDF, TEXT, another XML format etc.

Linkability, in-line useability, universally accepted standard with free/inexpensive tools etc

---

Q 93: What is the difference between a SAX parser and a DOM parser? SF PI MI

A 93:

SAX parser	DOM parser
A SAX (**S**imple **A**PI for **X**ML) parser does not create any internal structure. Instead, it takes the occurrences of components of an input document **as events** (i.e., **event driven**), and tells the client what it reads as it reads through the input document.	A DOM (**D**ocument **O**bject **M**odel) parser **creates a tree structure in memory** from an input document and then waits for requests from client.
A SAX parser serves the client application always only with pieces of the document at any given time.	A DOM parser always serves the client application with the entire document no matter how much is actually needed by the client.
A SAX parser, however, is much more space efficient in case of a big input document (because it creates no internal structure). What's more, it runs faster and is easier to learn than DOM parser because its API is really simple. But from the functionality point of view, it provides a fewer functions, which means that the users themselves have to take care of more, such as creating their own data structures.	A DOM parser is rich in functionality. It creates a DOM tree in memory and allows you to access any part of the document repeatedly and allows you to modify the DOM tree. But it is space inefficient when the document is huge, and it takes a little bit longer to learn how to work with it.
Use SAX parser when ▪ Input document is too big for available memory. ▪ When only a part of the document is to be read and we create the data structures of our own. ▪ If you use SAX, you are using much less memory and performing much less dynamic memory allocation.	Use DOM when ▪ Your application has to access various parts of the document and using your own structure is just as complicated as the DOM tree. ▪ Your application has to change the tree very frequently and data has to be stored for a significant amount of time.
SAX Parser example: Xerces, Crimson etc Use JAXP (**J**ava **A**PI for **X**ML **P**arsing) which enables applications to parse and transform XML documents independent of the particular XML parser. Code can be developed with one SAX parser in mind and later on can be changed to another SAX parser without changing the application code.	**DOM Parser example**: XercesDOM, SunDOM, OracleDOM etc. Use JAXP (**J**ava **A**PI for **X**ML **P**arsing) which enables applications to parse and transform XML documents independent of the particular XML parser. Code can be developed with one DOM parser in mind and later on can be changed to another DOM parser without changing the application code.

Q 94: Which is better to store data as elements or as attributes? SF

A 94: A question arising in the mind of XML/DTD designers is whether to model and encode certain information using an *element*, or alternatively, using an *attribute*. The answer to the above question is not clear-cut. But the general guideline is:

- **Using an element:** <book><title>Lord of the Rings</title>...</book>: If you consider the information in question to be part of the essential material that is being expressed or communicated in the XML, put it in an element
- **Using an attribute:** <book title=" Lord of the Rings "/>: If you consider the information to be peripheral or incidental to the main communication, or purely intended to help applications process the main communication, use attributes.

The principle is **data goes in elements** and **metadata goes in attributes.** Elements are also useful when they contain special characters like “<”, “>”, etc which are harder to use in attributes.

Q 95: What is XPATH? What is XSLT/XSL/XSL-FO/XSD/DTD etc? What is JAXB? What is JAXP? SF

A 95:

What is	Explanation	Example

XML	XML stands for eXtensible Markup Language	Sample.xml
		
		`<?xml version="1.0"?>`
		`<note>`
		`<to>Peter</to>`
		`<from>Paul</from>`
		`<title>Invite</title>`
		`<content language="English">Not Much</content>`
		`< content language="Spanish">No Mucho</content >`
		`</note>`

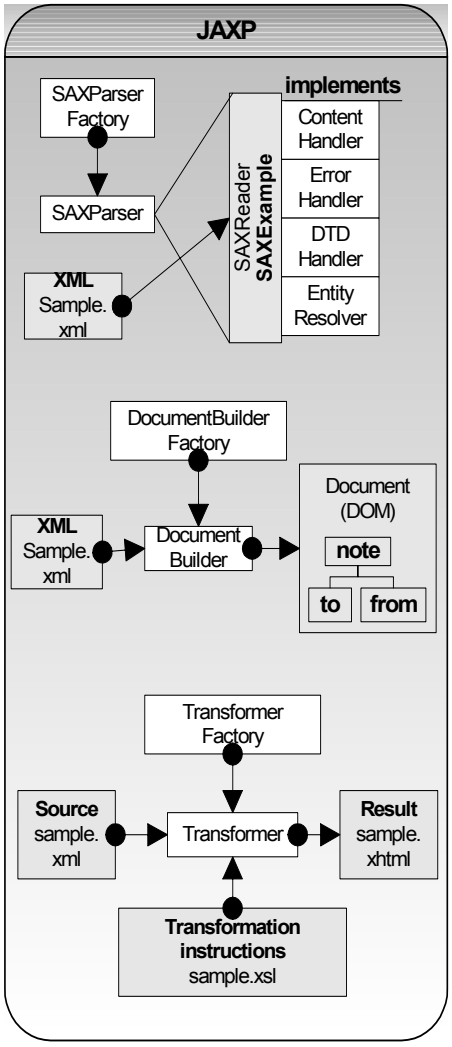
DTD	DTD stands for Document Type Definition. XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data. So the DTD is the building blocks or schema definition of the XML document.	Sample.dtd
		
		`<!ELEMENT note (to, from, title, content)>`
		`<!ELEMENT to (#PCDATA)>`
		`<!ELEMENT from (#PCDATA)>`
		`<!ELEMENT title (#PCDATA)>`
		`<!ELEMENT content (#PCDATA)>`
		`<!ATTLIST content language CDATA #Required>`

XSD	XSD stands for Xml Schema Definition, which is a successor of DTD. So XSD is a building block of an XML document.
	
	If you have DTD then why use XSD you may ask?
	
	XSD is more powerful and extensible than DTD. XSD has:
	
	• Support for simple and complex data types.
	• Uses XML syntax. So XSD are extensible just like XML because they are written in XML.
	• Better data communication with the help of data types. For example a date like 03-04-2005 will be interpreted in some countries as 3rd of April 2005 and in some other countries as 04th March 2005.
		Sample.xsd
		
		`<?xml version="1.0"?>`
		`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"`
		`targetNamespace="http://www.w3schools.com"`
		`xmlns="http://www.w3schools.com"`
		`elementFormDefault="qualified">`
		
		`<xs:element name="note">`
		`<xs:complexType>`
		`<xs:sequence>`
		`<xs:element name="to" type="xs:string"/>`
		`<xs:element name="from" type="xs:string"/>`
		`<xs:element name="title" type="xs:string"/>`
		`<xs:element name="content" type="xs:string"/>`
		`</xs:sequence>`
		`</xs:complexType>`
		`<xs:attribute name="language" type="xs:string"`
		`use="Required" />`
		`</xs:element>`
		
		`</xs:schema>`

XSL	XSL stands for eXtensible Stylesheet Language. The XSL consists of 3 parts:
	
	• XSLT: Language for transforming XML documents from one to another.
	
	• XPath: Language for defining the parts of an XML document.
	
	• XSL-FO: Language for formatting XML documents. For example to convert an XML document to a PDF document etc.
	
	XSL can be thought of as a set of languages that can :
	
	• Define parts of an XML
	• Transform an XML document to XHTML (eXtensible Hyper Text Markup Language) document.
	• Convert an XML document to a PDF document.
	• Filter and sort XML data.
	
	XSLT processor example: Xalan (From Apache)
	
	PDF Processor example: FOP (Formatting Objects Processor from Apache)
		To convert the Sample.xml file to a XHTML file let us apply the following Sample.xsl through XALAN parser.
		
		Sample.xsl
		
		`<?xml version="1.0"?>`
		`<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">`
		`<xsl:template match="/">`
		`<xsl:apply-templates select="note " />`
		`</xsl:template>`
		
		`<xsl:template match="note">`
		`<html>`
		`<head>`
		`<title><xsl:value-of`
		`select="content/@language">`
		`</title>`
		`</head>`
		`</html>`
		`</xsl:template>`
		`</xsl:stylesheet>`
		
		You get the following output XHTML file:
		
		Sample.xhtml
		
		`<html>`
		`<head>`
		`<title>English</title>`

		</head> </html> Now to convert the Sample.xml into a PDF file apply the following FO (Formatting Objects) file Through the FOP processor. Sample.fo <?xml version="1.0" encoding="ISO-8859-1"?> <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"> <fo:layout-master-set> <fo:simple-page-master master-name="A4"> </fo:simple-page-master> </fo:layout-master-set> <fo:page-sequence master-reference="A4"> <fo:flow flow-name="xsl-region-body"> <fo:block> <xsl:value-of select="content[@language='English']"> </fo:block> </fo:flow> </fo:page-sequence> </fo:root> which gives a basic Sample.pdf which has the following line Not Much
XPath	Xml Path Language, a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer. We can write both the patterns (context-free) and expressions using the XPATH Syntax. XPATH is also used in XQuery.	As per Sample.xsl <xsl:template match="content[@language='English']"> …… <td><xsl:value-of select="content/@language" /></td>

JAXP	Stands for Java API for XML Processing. This provides a common interface for creating and using SAX, DOM, and XSLT APIs in Java regardless of which vendor's implementation is actually being used (just like the JDBC, JNDI interfaces). JAXP has the following packages:	DOM example using JAXP:
		
		DocumentBuilderFactory dbf =
		DocumentBuilderFactory.newInstance();
		DocumentBuilder db = dbf.newDocumentBuilder();
		Document doc =
		db.parse(new File("xml/Test.xml"));
		NodeList nl = doc.getElementsByTagName("to");
		Node n = nl.item(0);
		System.out.println(n.getFirstChild().getNodeValue());
		
		SAX example using JAXP:
		
		SAXParserFactory spf =
		SAXParserFactory.newInstance();
		SAXParser sp = spf.newSAXParser();
		SAXExample se = new SAXExample();
		sp.parse(new File("xml/Sample.xml"),se);
		
		where SAXExample.Java code snippet
		
		public class SAXExample extends DefaultHandler {
		
		public void startElement(
		String uri,
		String localName,
		String qName,
		Attributes attr)
		throws SAXException {
		
		System.out.println("--->" + qName);
		}
		...
		
		}

<table>
<tr>
<td></td>
<td>


javax.xml.parsers → common interface for different vendors of SAX, DOM parsers).
org.xml.sax → Defines basic SAX API.
org.w3c.dom → Defines Document Object Model and its componenets.
javax.xml.transform → Defines the XSLT API which allows you to transform XML into other forms like PDF, XHTML etc.

Required JAR files are jaxp.jar, dom.jar, xalan.jar, xercesImpl.jar.
</td>
<td>
The DefaultHandler implements ContentHandler, DTDHandler, EntityResolver, ErrorHandler

XSLT example using JAXP:

<pre>StreamSource xml =
   new StreamSource(new File("/xml/Sample.xml"));
StreamSource xsl = new StreamSource(
   new File("xml/Sample.xsl"));
StreamResult result =
   new StreamResult(new File("xml/Sample.xhtml"));

TransformerFactory tf =
                TransformerFactory.newInstance();
Transformer t = tf.newTransformer(xsl);
t.transform(xml, result);</pre>
This gives you Sample.xhtml

<pre><html>
  <head>
    <title>English</title>
  </head>
</html></pre>
</td>
</tr>
<tr>
<td>JAXB</td>
<td>Stands for Java API for XML Binding. This standard defines a mechanism for writing out Java objects as XML (marshalling) and for creating Java objects from XMLstructures (unmarshalling). (You compile a class description to create the Java classes, and use those classes in your application.)</td>
<td>
Lets look at some code:

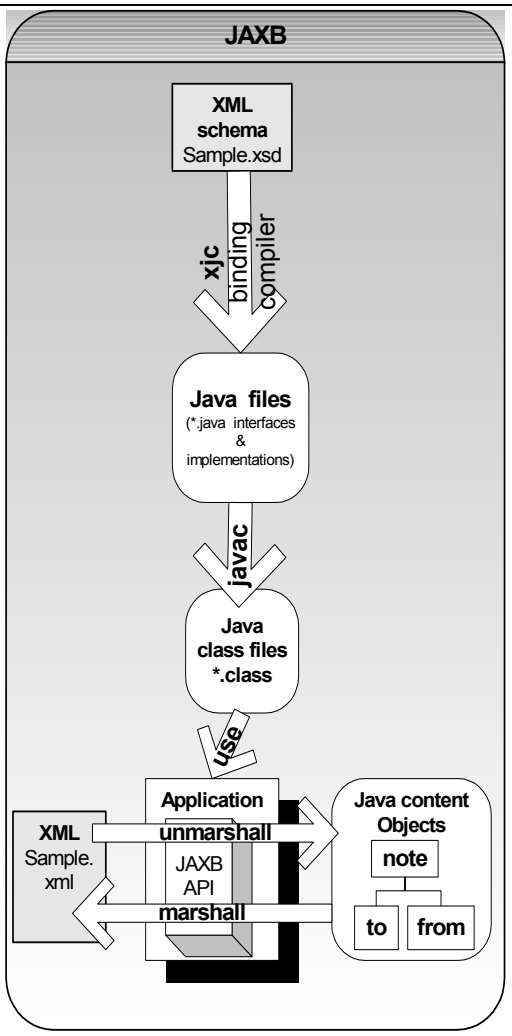
For binding:

<pre>xjc.sh –p com.binding sample.xsd –d work</pre>
-p identifies the package for the generated Java files (ie *.Java)

-d option identifies the target.

Unmarshalling the XML document:

<pre>JAXBContext jc = JAXBContext.newInstance(
                                “com.binding”);
Unmarshaller um = jc.createUnmarshaller();
Object o = um.unMarshall(
                        new File(“/xml/”));</pre>
</td>
</tr>
</table>




```
Note n = (Note) n;
System.out.println(n.getFrom().getContent().get(0));
System.out.println(n.getTo().getContent().get(0));
```

Now to modify the in memory object content:

```
n. getFrom().getContent().set(0, “newValue”);
```

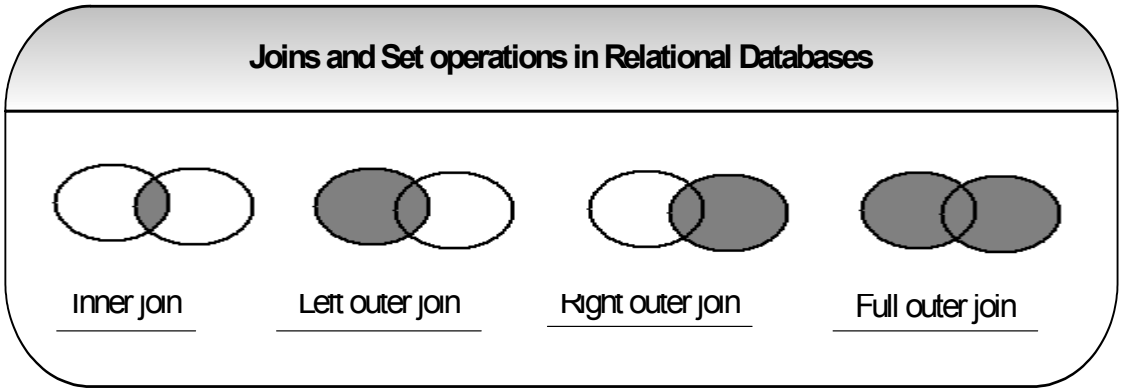
Marshalling the change back to different XML file:

```
Marshaller m = jc.createMarshaller();

FileOutputStream fos = new FileOutputStream(
                new File(“/xml/SampleNew.xml”));

m.marshall(n, fos);
```

Enterprise – SQL, Tuning and O/R mapping

Q 96: Explain inner and outer joins? SF

A 96: Joins allow database users to combine data from one table with data from one or more other tables (or views, or synonyms). Tables are joined two at a time making a new table containing all possible combinations of rows from the original two tables. Lets take an example (syntax vary among RDBMS):



Employees table

Id	firstname	Surname	state
1001	John	Darcy	NSW
1002	Peter	Smith	NSW
1003	Paul	Gregor	NSW
1004	Sam	Darcy	VIC

Executives table

Id	firstname	Surname	state
1001	John	Darcy	NSW
1002	Peter	Smith	NSW
1005	John	Gregor	WA

Inner joins: Chooses the join criteria using any column names that happen to match between the two tables. The example below displays only the employees who are executives as well.

```
SELECT emp.firstname, exec.surname FROM employees emp, executives exec
WHERE emp.id = exec.id;
```

The output is:

Firstname	surname
John	Darcy
Peter	Smith

Left Outer joins: A problem with the inner join is that only rows that match between tables are returned. The example below will show all the employees and fill the null data for the executives.

```
SELECT emp.firstname, exec.surname FROM employees emp left join executives exec
ON emp.id  = exec.id;
```

On oracle

```
SELECT emp.firstname, exec.surname FROM employees emp, executives exec
    where emp.id  = exec.id(+);
```

The output is:

Firstname	surname
John	Darcy
Peter	Smith
Paul	
Sam	

Right Outer join: A problem with the inner join is that only rows that match between tables are returned. The example below will show all the executives and fill the null data for the employees.

```
SELECT emp.firstname, exec.surname FROM employees emp right join executives exec
    ON emp.id  = exec.id;
```

On oracle

```
SELECT emp.firstname, exec.surname FROM employees emp, executives exec
    WHERE emp.id(+)  = exec.id;
```

The output is:

Firstname	surname
John	Darcy
Peter	Smith
	Gregor

Full outer join: To cause SQL to create both sides of the join

```
SELECT emp.firstname, exec.surname FROM employees emp full join executives exec
ON emp.id  = exec.id;
```

On oracle

```
SELECT emp.firstname, exec.surname FROM employees emp, executives exec
    WHERE emp.id = exec.id (+)
```

```
UNION

SELECT emp.firstname, exec.surname FROM employees emp, executives exec
      WHERE emp.id(+) = exec.id
```

Note: Oracle9i introduced the ANSI compliant join syntax. This new join syntax uses the new keywords inner join, left outer join, right outer join, and full outer join, instead of the (+) operator.

The output is:

Firstname	surname
John	Darcy
Paul	
Peter	Smith
Sam	
	Gregor

Self join: A self-join is a join of a table to itself. If you want to find out all the employees who live in the same city as employees whose first name starts with “Peter”, then one way is to use a sub-query as shown below:

```
SELECT emp.firstname, emp.surname FROM employees emp WHERE
  city IN (SELECT city FROM employees where firstname like ‘Peter’)
```

The sub-queries can degrade performance. So alternatively we can use a self-join to achieve the same results.

On oracle

```
SELECT emp.firstname, emp.surname FROM employees emp, employees emp2
      WHERE emp.state  = emp2.state
      AND emp2.firstname LIKE 'Peter'
```

The output is:

Firstname	Surname
John	Darcy
Peter	Smith
Paul	Gregor

Q 97: Explain a sub-query? How does a sub-query impact on performance? **SF** **PI**

A 97: It is possible to embed a SQL statement within another. When this is done on the **WHERE** or the **HAVING** statements, we have a subquery construct. What is subquery useful for? It is used to join tables and there are cases where the only way to correlate two tables is through a subquery.

```
SELECT emp.firstname, emp.surname FROM employees emp WHERE
emp.id NOT IN (SELECT id FROM executives);
```

There are performance problems with sub-queries, which may return NULL values. The above sub-query can be re-written as shown below by invoking a **correlated sub-query**:

```
SELECT emp.firstname, emp.surname FROM employees emp WHERE
emp.id NOT EXISTS (SELECT id FROM executives);
```

The above query can be re-written as an outer join for a faster performance as shown below:

```
SELECT emp.firstname, exec.surname FROM employees emp left join executives exec
on emp.id  = exec.id AND exec.id IS NULL;
```

The above execution plan will be faster by eliminating the sub-query.

Q 98: What is normalization? When to denormalize? **DC** **PI**

A 98: Normalization is a design technique that is widely used as a guide in designing relational databases. Normalization is essentially a two step process that puts data into tabular form by removing repeating groups and then removes duplicated data from the relational tables (Additional reading recommended).

Redundant data wastes disk space and creates maintenance problems. If data that exists in more than one place must be changed, the data must be changed in exactly the same way in all locations which is time consuming and prone to errors. A change to a customer address is much easier to do if that data is stored only in the Customers table and nowhere else in the database.

Inconsistent dependency is a database design that makes certain assumptions about the location of data. For example, while it is intuitive for a user to look in the Customers table for the address of a particular customer, it may not make sense to look there for the salary of the employee who calls on that customer. The employee's salary is related to, or dependent on, the employee and thus should be moved to the Employees table. Inconsistent dependencies can make data difficult to access because the path to find the data may not be logical, or may be missing or broken.

First normal form	Second Normal Form	Third Normal Form
A database is said to be in First Normal Form when all entities have a unique identifier or key, and when every column in every table contains only a single value and doesn't contain a repeating group or composite field.	A database is in Second Normal Form when it is in First Normal Form plus every non-primary key column in the table must depend on the entire primary key, not just part of it, assuming that the primary key is made up of composite columns.	A database is in Third Normal Form when it is in Second Normal Form and each column that isn't part of the primary key doesn't depend on another column that isn't part of the primary key.

When to denormalize? Normalize for accuracy and denormalize for performance.

Typically, transactional databases are highly normalized. This means that redundant data is eliminated and replaced with keys in a one-to-many relationship. Data that is highly normalized is constrained by the primary key/foreign key relationship, and thus has a high degree of *data integrity*. Denormalized data, on the other hand, creates redundancies; this means that it's possible for denormalized data to lose track of some of the relationships between atomic data items. However, since all the data for a query is (usually) stored in a single row in the table, it is *much* faster to retrieve.

Q 99: How do you implement one-to-one, one-to-many and many-to-many relationships while designing tables? **SF**

A 99: **One-to-One relationship** can be implemented as a single table and rarely as two tables with primary and foreign key relationships.

One-to-Many relationships are implemented by splitting the data into two tables with primary key and foreign key relationships.

Many-to-Many relationships are implemented using join table with the keys from both the tables forming the composite primary key of the junction table.

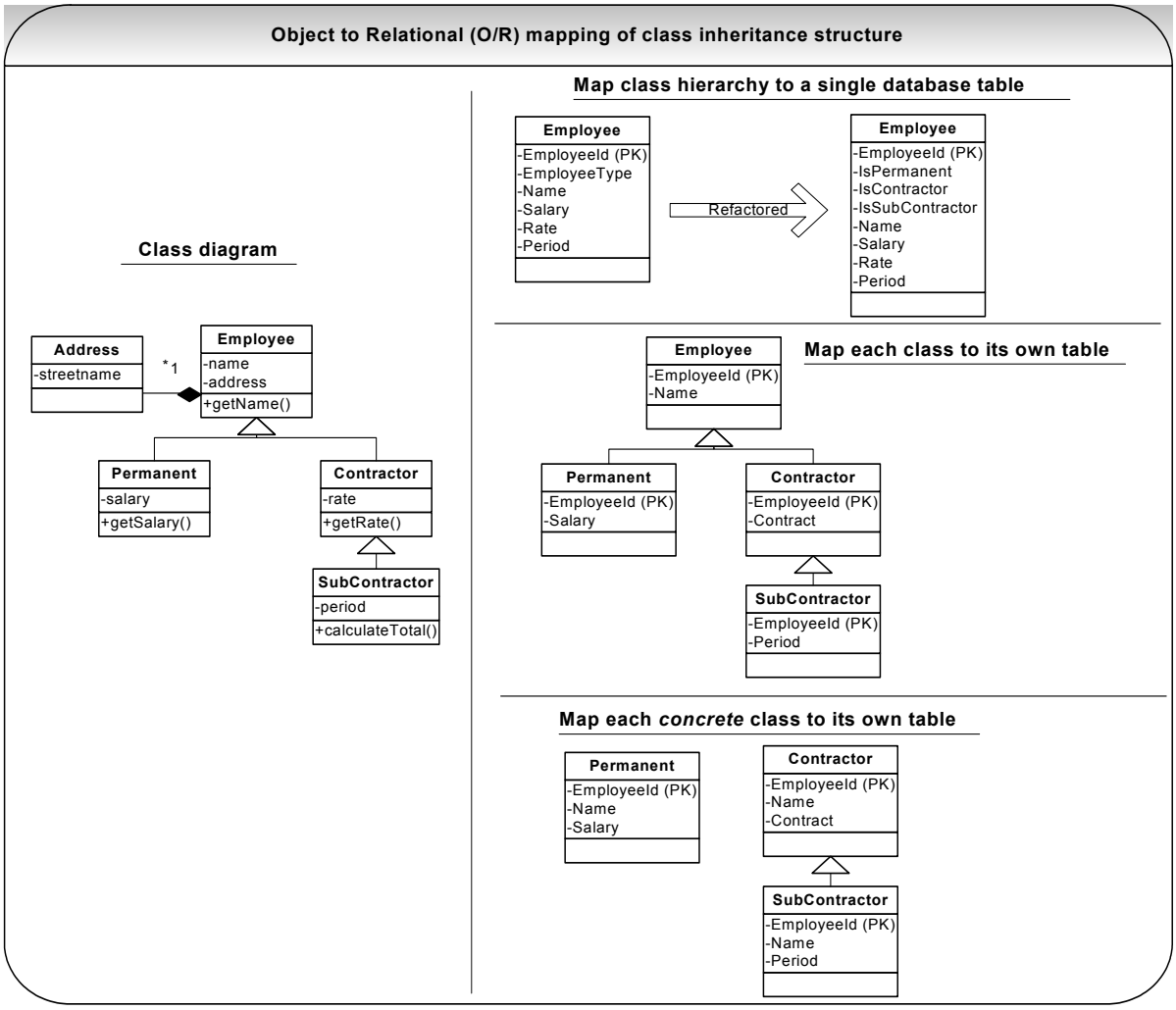
Q 100: How can you performance tune your database? **PI**

A 100:

- Denormalize your tables where appropriate.
- Proper use of index columns: An index based on numeric fields is more efficient than an index based on character columns.
- Reduce the number of columns that make up a composite key.
- Proper partitioning of tablespaces and create a special tablespace for special data types like CLOB, BLOB etc.
- Data access performance can be tuned by using stored procedures to crunch data in the database server to reduce the network overhead and also caching data within your application to reduce the number of accesses.

Q 101: How will you map objects to a relational database? How will you map class inheritance to relational data model? **DC**

A 101: Due to impedance mismatch between object and relational technology you need to understand the process of mapping classes (objects) and their relationships to tables and relationships between them in a database. Classes represent both behaviour and data whereas relational database tables just implement data. Database schemas have keys (primary keys to uniquely identify rows and foreign keys to maintain relationships between rows) whereas object schema does not have keys and instead use references to implement relationships to other objects. Let us look at some basic points on mapping:



- Classes map to tables in a way but not always directly.
- An attribute of a class can be mapped to zero or more columns in a database. Not all attributes are persistent.
- Some attributes of an object are objects itself. For example an *Employee* object has an Address object as an attribute. This is basically an **association** relationship between two objects (i.e. *Employee* and *Address*). This is a recursive relationship where at some point the attribute will be mapped to zero or more columns. In this example attributes of the *Address* class will be mapped zero or more columns.

- In its simple form an attribute maps to a single column whereas each has same type (ie attribute is a string and column is a char, or both are dates etc). When you implement mapping with different types (attribute is a currency and column is a float) then you will need to be able to convert them back and forth.

How do you map inheritance class structure to relational data model?

Relational databases do not support inheritance. Class inheritance can be mapped to relational tables as follows:

Map class hierarchy to single database table: The whole class hierarchy can be stored in a single table by adding an additional column named “EmployeeType”. The column “EmployeeType” will hold the values “Permanent”, “Contract” and “SubContract”. New employee types can be added as required. Although this approach is straightforward it tends to break when you have combinations like an employee is of type both “Contractor” and “SubContractor”. So when you have combinations, you can use refactored table by replacing type code column “EmployeeType” with boolean values such as isPermanent, isContractor and isSubContractor.

Map each class to its own table: You create one table per class. The data for a permanent employee is stored in two tables (Employee and Permanent), therefore to retrieve this data you need to join these two tables. To support additional employee type say a *Contractor*, add a new table.

Map each concrete class to its own table: You create one table per concrete class. There are tables corresponding to each class like Permanent, Contractor and SubContractor. So join is not required. To support additional employee type, add a new table.

So which approach to use? Easiest approach is to have one table per hierarchy and easy to refactor. If you need a "pure design approach" then use one table per class approach. Try to stay away from one table per concrete class approach because it makes refactoring difficult by copying data back and forth between tables. No approach is ideal for all situations.

Another option for mapping inheritance into relational database is to take a generic meta-data driven approach. This approach supports all forms of mapping. In this approach, value of a single attribute will be stored as a row in a table called "Value". So, to store 5 attributes you need 5 rows in "Value" table. You will have a table called "Class" where class names are stored, a table called "Inheritance" where subclass and superclass information is stored, a table called "Attributes" where class attributes are stored and an "AttributeType" lookup table.

Q 102: What is a view? Why will you use a view? What is an aggregate function? Etc. SF PI

A 102:

Question	Explanation
What is view? Why use a view?	View is a precompiled SQL query, which is used to select data from one or more tables. A view is like a table but it doesn't physically take any space (ie not materialised). Views are used for ▪ Providing inherent security by exposing only the data that is needed to be shown to the end user. ▪ Enabling re-use of SQL statements. ▪ Allows changes to the underlying tables to be hidden from clients, aiding maintenance of the database schema (i.e. encapsulation). Views with multiple joins and filters can dramatically degrade performance because views contain no data and any retrieval needs to be processed. The solution for this is to use materialised views or create de-normalised tables to store data. This technique is quite handy in overnight batch processes where a large chunk of data needs to be processed. Normalised data can be read and inserted into some temporary de-normalised table and processed with efficiency.
Explain aggregate SQL functions?	SQL provides aggregate functions to assist with the summarisation of large volumes of data. We'll look at functions that allow us to add and average data, count records meeting specific criteria and find the largest and smallest values in a table. **ORDERID FIRSTNAME SURNAME QTY UNITPRICE** 1001 John Darcy 25 10.5 1002 Peter Smith 25 10.5 1003 Sam Gregory 25 10.5 SELECT **SUM**(QTY) AS Total FROM Orders; *The output is:* Total = 75 SELECT **AVG**(UnitPrice * QTY) As AveragePrice FROM Orders; *The output is:* AveragePrice = 262.50 If we inserted another row to the above table: **ORDERID FIRSTNAME SURNAME QTY UNITPRICE** 1004 John Darcy 20 10.50 SELECT FIRSTNAME,SUM(QTY) FROM orders **GROUP BY** FIRSTNAME **HAVING** SUM(QTY)>25; *The output is:* John 45
Explain INSERT, UPDATE, and DELETE statements?	**INSERT** statements can be carried out several ways: **INSERT** INTO ORDERS values (1004, 'John', 'Darcy', 20, 10.50);

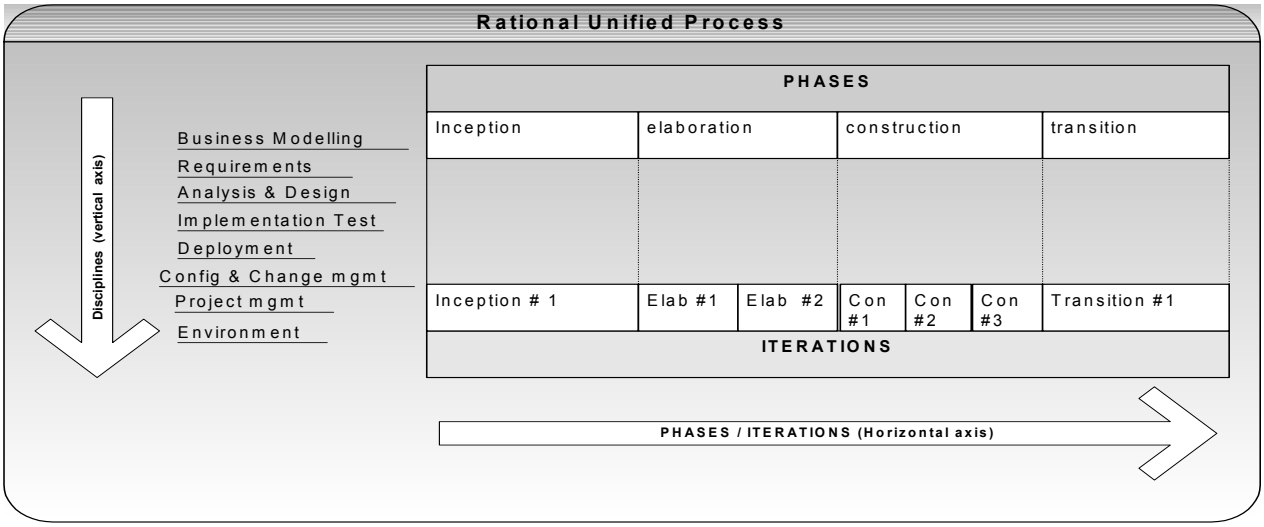
	The above statement is fine but the one below is recommended since it is less ambiguous and less prone to errors. **INSERT** INTO ORDERS (orderid, firstname, surname, qty, unitprice) values (1005, 'John', 'Darcy', 20, 10.50); We can also use INSERT with the SELECT statements as shown below **INSERT** into NEW_ORDERS (orderid, firstname, surname, qty, unitprice) SELECT orderid, firstname, surname, qty, unitprice FROM orders WHERE orderid = 1004; **UPDATE** statement allows you to update a single or multiple statements. **UPDATE** ORDERS set firstname='Peter', surname='Piper' WHERE orderid=1004; Also can have more complex updates like **UPDATE** supplier SET supplier_name = (SELECT customer.name FROM customers WHERE customers.customer_id = supplier.supplier_id) WHERE EXISTS (SELECT customer.name FROM customers WHERE customers.customer_id = supplier.supplier_id); **DELETE** statements allow you to remove records from the database. DELETE FROM ORDERS WHERE orderid=1004; We can clear the entire table using **TRUNCATE** TABLE employees; When running UPDATE/DELETE care should be taken to include WHERE clause otherwise you can inadvertently modify or delete records which you do not intend to UPDATE/DELETE.
How can you compare a part of the name rather than the entire name?	You can use wild card characters like: • * (% in oracle) → Match any number of characters. • ? (_ in oracle) → Match a single character. **To find all the employees who has "au":** SELECT * FROM employees emp WHERE emp.firstname LIKE '%au%';
How do you get distinct entries from a table?	The SELECT statement in conjunction with **DISTINCT** lets you select a set of distinct values from a table in a database. SELECT **DISTINCT** empname FROM emptable
How can you find the total number of records in a table?	Use the **COUNT** key word: SELECT **COUNT**(*) FROM emp WHERE age>25
What's the difference between a primary key and a unique key?	Both primary key and unique key enforce uniqueness of the column on which they are defined. But by default primary key creates a **clustered index** on the column, whereas unique creates a **non-clustered index** by default. Another major difference is that, primary key doesn't allow NULLs, but unique key allows one NULL only.
What are constraints? Explain different types of constraints.	Constraints enable the RDBMS enforce the integrity of the database automatically, without needing you to create triggers, rule or defaults. Types of constraints: **NOT NULL**, **CHECK**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY**
What is an index? What are the types of indexes? How many clustered indexes can be created on a table? What are the	The books you read have indexes, which help you to go to a specific key word faster. The database indexes are similar. Indexes are of two types. **Clustered indexes** and non-clustered indexes. When you

advantages and disadvantages of creating separate index on each column of a table?	create a clustered index on a table, all the rows in the table are stored in the order of the clustered index key. So, there can be only one clustered index per table. **Non-clustered indexes** have their own storage separate from the table data storage. The row located could be the RowID or the clustered index key, depending up on the absence or presence of clustered index on the table. If you create an index on each column of a table, it improves the query performance, as the query optimizer can choose from all the existing indexes to come up with an efficient execution plan. At the same time, data modification operations (such as INSERT, UPDATE, and DELETE) will become slow, as every time data changes in the table, all the indexes need to be updated. Another disadvantage is that, indexes need disk space, the more indexes you have, more disk space is used.

Enterprise - RUP & UML

Q 103: What is RUP? SD

A 103: Rational Unified Process (RUP) is a general framework that can be used to describe a development process. The software development cycle has got 4 phases in the following order **I**nception, **E**laboration, **C**onstruction, and **T**ransition.



The core of the phases is **state-based**, and the state is determined by what fundamental questions you are trying to answer:

- **Inception** - do you and the customer have a shared understanding of the system?
- **Elaboration** - do you have baseline architecture to be able to build the system?
- **Construction** - are you developing a product?
- **Transition** - are you trying to get the customer to take ownership of the system?

RUP is based on a few important philosophies and principles:

- A software project team should **plan ahead**.
- It should know **where it is going**.
- It should capture project knowledge in a **storable** and **extensible** form.

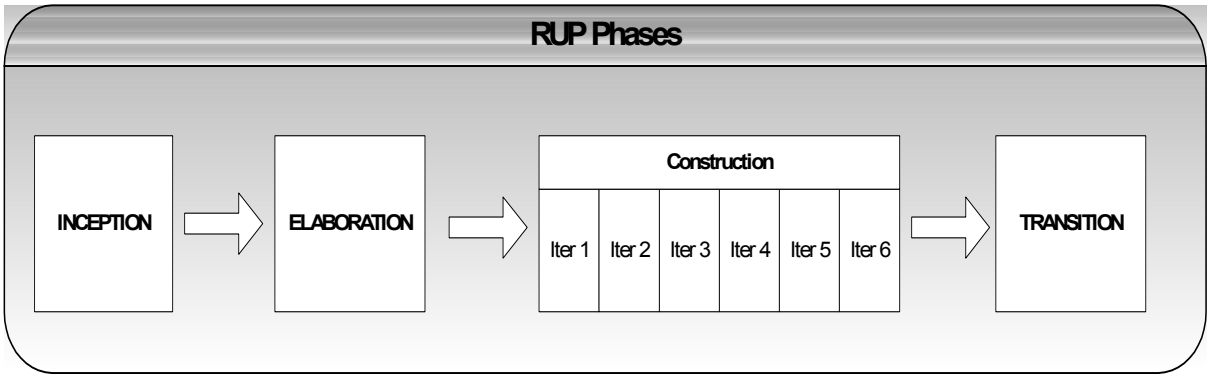
The best practices of RUP involve the following major 5 properties:

Best practice property	Description
Use case driven	Interaction between the users and the system.
Architecture centric	Based on architecture with clear relationships between architectural components.
Iterative	The problem and the solution are divided into more manageable smaller pieces, where each

	iteration will be addressing one of those pieces.
Incremental	Each iteration builds incrementally on the foundation built in the previous iteration.
Controlled	Control with respect to process means you always know what to do next; control with respect to management means that all deliverables, artifacts, and code are under configuration management.

Q 104: Explain the 4 phases of RUP? **SD**

A 104:



- **Inception**: During the inception phase, you work out the business case for the project. You also will be making a rough cost estimate and return on investment. You should also outline the scope and size of the project.

  The fundamental question you ask at the end of this phase: do you and the customer have a shared understanding of the system?

- **Elaboration**: At this stage you have the go ahead of the project however only have vague requirements. So at this stage you need to get a better understanding of the problem. Some of the steps involved are:

  - What is it you are actually going to build?
  - How are you going to build it?
  - What technology are you going to use?
  - Analysing and dealing with requirement risks, technological risks, skill risks, political risks etc.
  - Develop a **domain model**, **use case model** and **a design model.** The UML techniques can be used for the model diagrams (e.g. class diagrams, sequence diagrams etc).

  An important result of the elaboration phase is that you have a **baseline architecture**. This architecture consists of:

  - A list of use cases depicting the requirements.
  - The domain model, which captures your understanding of the domain with the help of UML class diagrams.
  - Selection of key implementation technology and how they fit together. For example: Java/J2EE with JSP, Struts, EJB, XML, etc.

  The fundamental question you ask at the end of this phase: do you have a baseline architecture to be able to build the system?

- **Construction:** In this phase you will be building the system in a series of iterations. Each iteration is a mini project. You will be performing analysis, design, unit testing, coding, system testing, and integration testing for the use cases assigned to each iteration. The iterations within the construction phase are incremental and iterative. Each iteration builds on the use cases developed in the previous iterations. The each iteration will involve code rewrite, refactoring, use of design patterns etc.

  The basic documentation required during the construction phase is:

  - A **class diagram** and a **sequence diagram**.
  - Some text to pull the diagrams together.

- If a class has complex life cycle behaviour then a **state diagram** is required.
- If a class has a complex computation then an **activity diagram** is required.

The fundamental question you ask at the end of this phase: do you have a developed product?

- **Transition:** During this phase you will be delivering the finished code regularly. During this phase there is no coding to add functionality unless it is small and essential. There will be bug fixes, code optimisation etc during this phase. An example of a transition phase is that the time between the beta release and the final release of a product.

  The fundamental question you ask at the end of this phase: are you trying to get the customer to take ownership of the developed product or system?

Q 105: What are the characteristics of RUP? Where can you use RUP? **SD**

A 105:

1. RUP is based on a few important philosophies and principles like planning ahead, knowing where the process is heading and capturing the project in storable and extensible manner.
2. It is largely based on OO analysis and design, and use case driven etc.
3. Iterative and incremental development as oppose to waterfall approach, which hides problems.
4. Architecture centric approach.

RUP is more suited for larger teams of 50-100 people. RUP can also be used as an agile (i.e. lightweight) process for smaller teams of 20-30 people, or as a heavy weight process for larger teams of 50-100 people. Extreme Programming (XP) can be considered as a subset of RUP. At the time of writing, the **agile (i.e lightweight) software development process** is gaining popularity and momentum across organizations. Several methodologies fit under this agile development methodology banner. All these methodologies share many characteristics like **iterative** and **incremental development**, **test driven development**, **stand up meetings to improve communication**, **automatic testing, build and continuous integration of code** etc. Refer **Q136** in Enterprise Java section.

Q 106: Why is UML important? **SD** **DC**

A 106: The more complicated the underlying system, the more critical the communication among everyone involved in developing and deploying the software. UML is a software blueprint language for analysts, designers and developers. UML provides a common vocabulary for the business analysts, architects, developers etc.

UML is applicable to the Object Oriented problem solving. UML begins with a **model**; A **model** is an abstraction of the underlying problem. The **domain** is the actual world from which the problem comes. The model consists of **objects**. The objects interact with each other by sending and receiving **messages**. The objects are characterised by **attributes** and **operations** (behaviours). The values of an object's attributes determine its **state**. The **classes** are the blueprints (or like templates) for objects. A class wraps **attributes** and **methods** into a single distinct entity. The objects are the **instances** of classes.

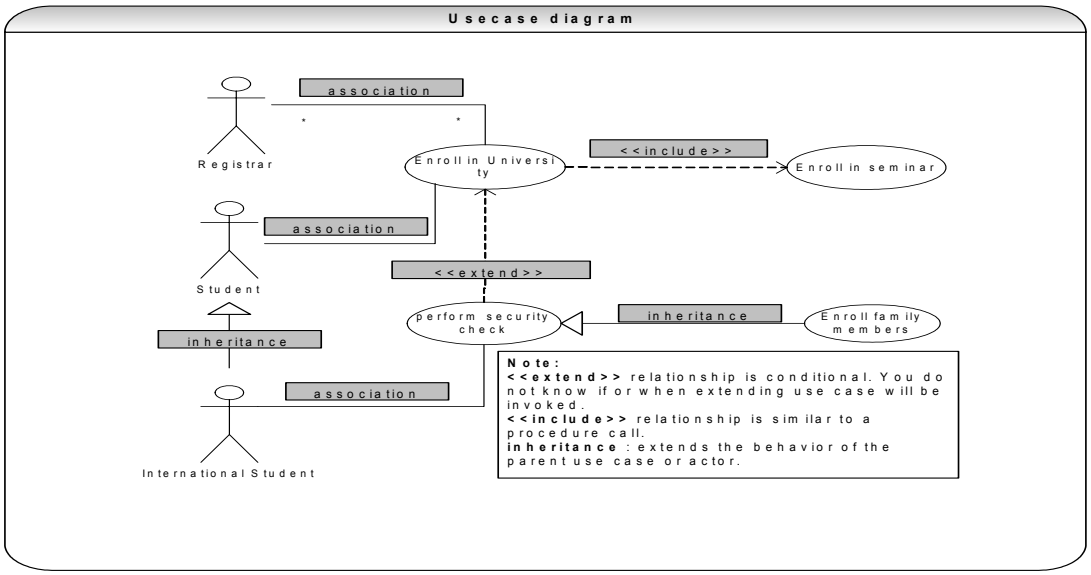
Q 107: What are the different types of UML diagrams? **SD** **DC**

A 107: **Use case diagrams**: Depicts the typical interaction between external users (actors) and the system. The emphasis is on **what** a system does rather than **how** it does it. A use case is a summary of scenarios for a single task or goal. An actor is responsible for initiating a task. The connection between actor and use case is a **communication association**.

Capturing use cases is one of the primary tasks of the **elaboration phase** of RUP. In its simplest usage, you capture a use case by talking to your users and discussing the various things they might want to do with the system.

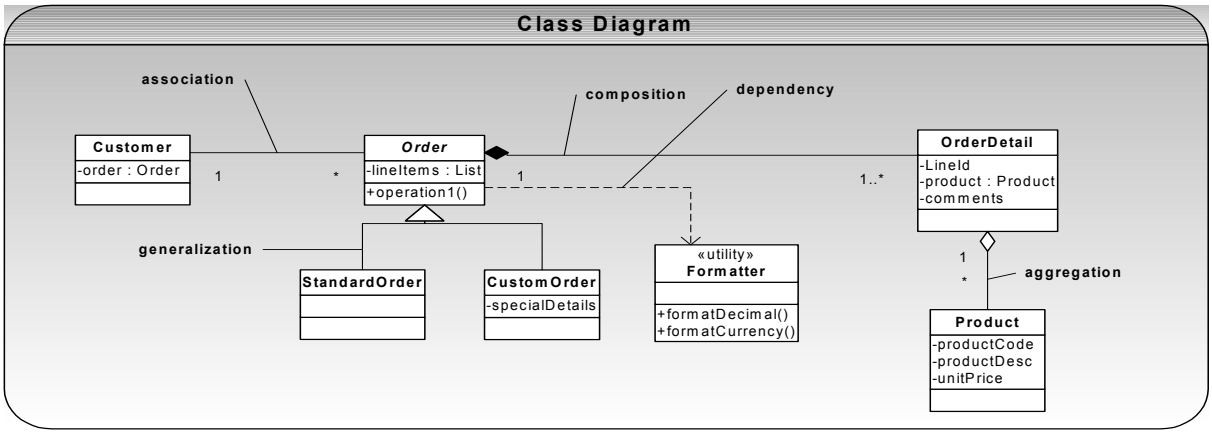
When to use 'use case' diagrams?

- Determining user requirements. New use cases often generate new requirements.
- Communicating with clients. The simplicity of the diagram makes use case diagrams a good way for designers and developers to communicate with clients.
- Generating test cases. Each scenario for the use case may suggest a suite of test cases.



Class diagrams: Class diagram technique is vital within **O**bject **O**riented methods. Class diagrams describe the types of objects in the system and the various static relationships among them. Class diagrams also show the attributes and the methods. Class diagrams have the following possible relationships:

- **Association**: A relationship between instances of 2 classes.
- **Aggregation**: An **association** in which one class belongs to a collection (does not always have to be a collection. You can also have cardinality of "1"). This is a **part of a whole** relationship where the **part** can exist without the **whole**. **For example**: A line item is whole and the products are the parts. If a line item is deleted then the products **need not be deleted**.
- **Composition**: An **association** in which one class belongs to a collection (does not always have to be a collection. You can also have cardinality of "1"). This is a **part of a whole** relationship where the **part** cannot exist without the **whole**. If the whole is deleted then the parts are deleted. **For example:** An Order is a whole and the line items are the parts. If an order is deleted then all the line items **should be deleted** as well (ie cascade deletes).
- **Generalization**: An inheritance link indicating that one class is a superclass of the other. The Generalization expresses the "**is a**" relationship whereas the association, aggregation and composition express the "**has a**" relationship.
- **Dependency**: A dependency is a weak relationship where one class requires another class. The dependency expresses the "**uses**" relationship. **For example:** A domain model class uses a utility class like Formatter etc.

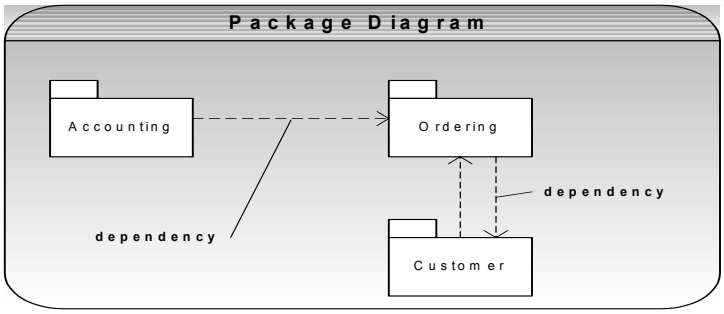


When to use class diagrams?

- Class diagrams are the backbone of **O**bject **O**riented methods. So they are used frequently.

- Class diagrams can have a conceptual perspective and an implementation perspective. During the analysis draw the conceptual model and during implementation draw the implementation model.

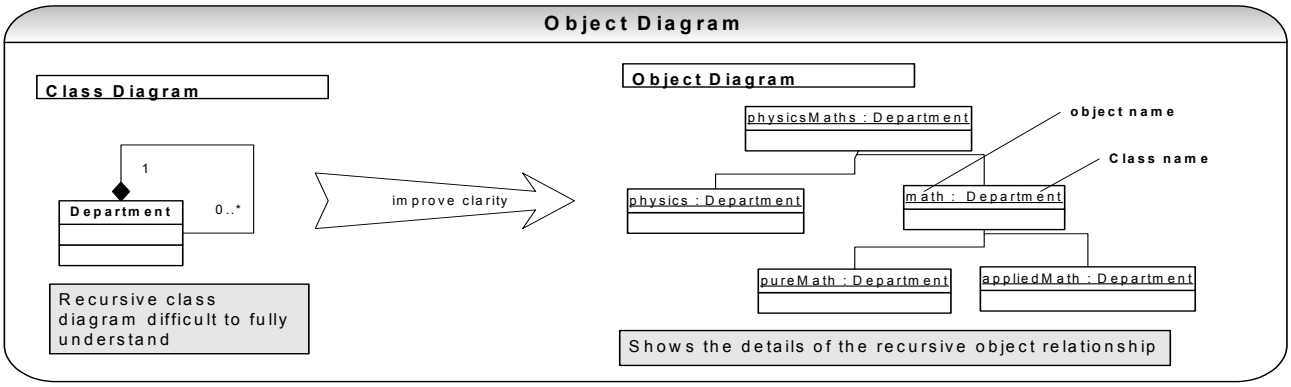
Package diagrams: To simplify complex class diagrams you can group classes into **packages**.



When to use package diagrams?

- Package diagrams are vital for large projects.

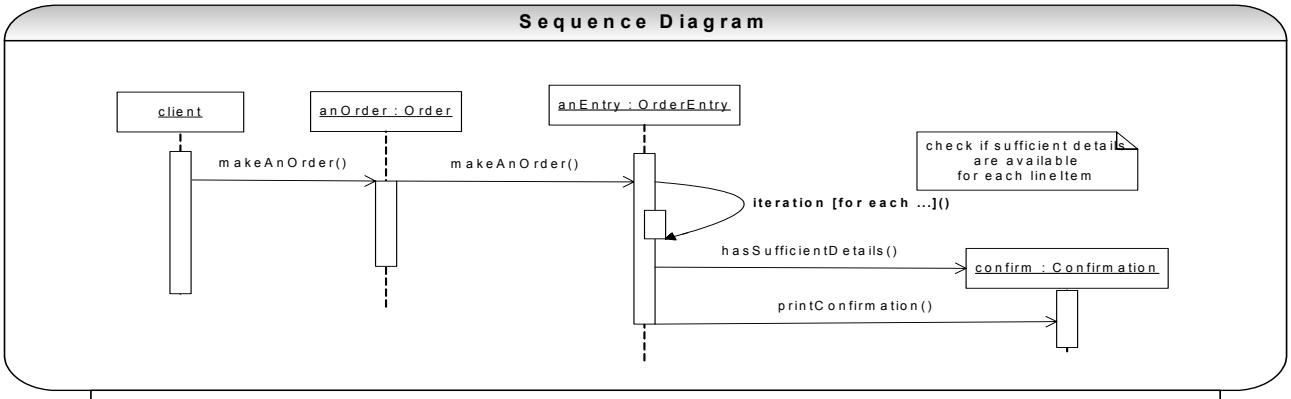
Object diagrams: Object diagrams show instances instead of classes. They are useful for explaining some complicated objects in detail about their recursive relationships etc.



When to use object diagrams?

- Object diagrams are a vital for large projects.
- They are useful for explaining structural relationships in detail for complex objects.

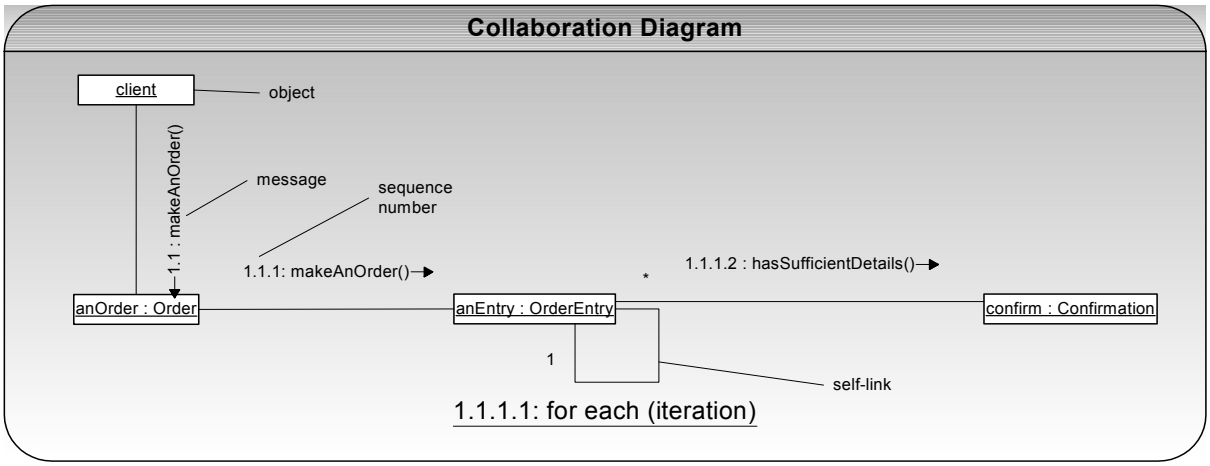
Sequence diagrams: Sequence diagrams are interaction diagrams which detail **what** messages are sent and **when**. The sequence diagrams are organized according to time. The time progresses as you move from top to bottom of the diagram. The objects involved in the diagram are shown from left to right according to **when** they take part.



Note: Each vertical dotted line is a **life line**. Each arrow is a **message**. The rectangular boxes on the life line are called the **activation bar** which represents the duration of execution of message.

Collaboration diagrams: Collaboration diagrams are also **interaction diagrams**. Collaboration diagrams convey the same message as the sequence diagrams. But the collaboration diagrams focus on the **object roles** instead of the **times** at which the messages are sent.

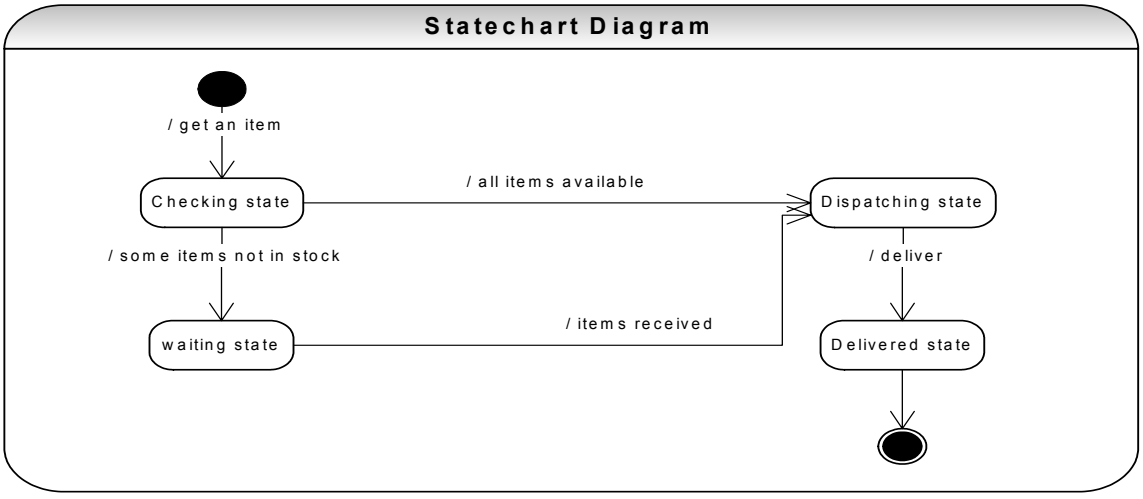
The collaboration diagrams use the decimal sequence numbers as shown in the diagram below to make it clear which operation is calling which other operation, although it can be harder to see the overall sequence. The top-level message is numbered 1. The messages at the same level have the same decimal prefix but different suffixes of 1, 2 etc according to when they occur.



When to use interaction diagrams?

- When you want to look at behaviour of several objects within a single use case. If you want to look at a single object across multiple use cases then use statechart diagram as described below.

State chart diagrams: Objects have behaviour and state. The state of an object is depends on its current activity or condition. This diagram shows the possible states of the object and the transitions that cause a change in its state.

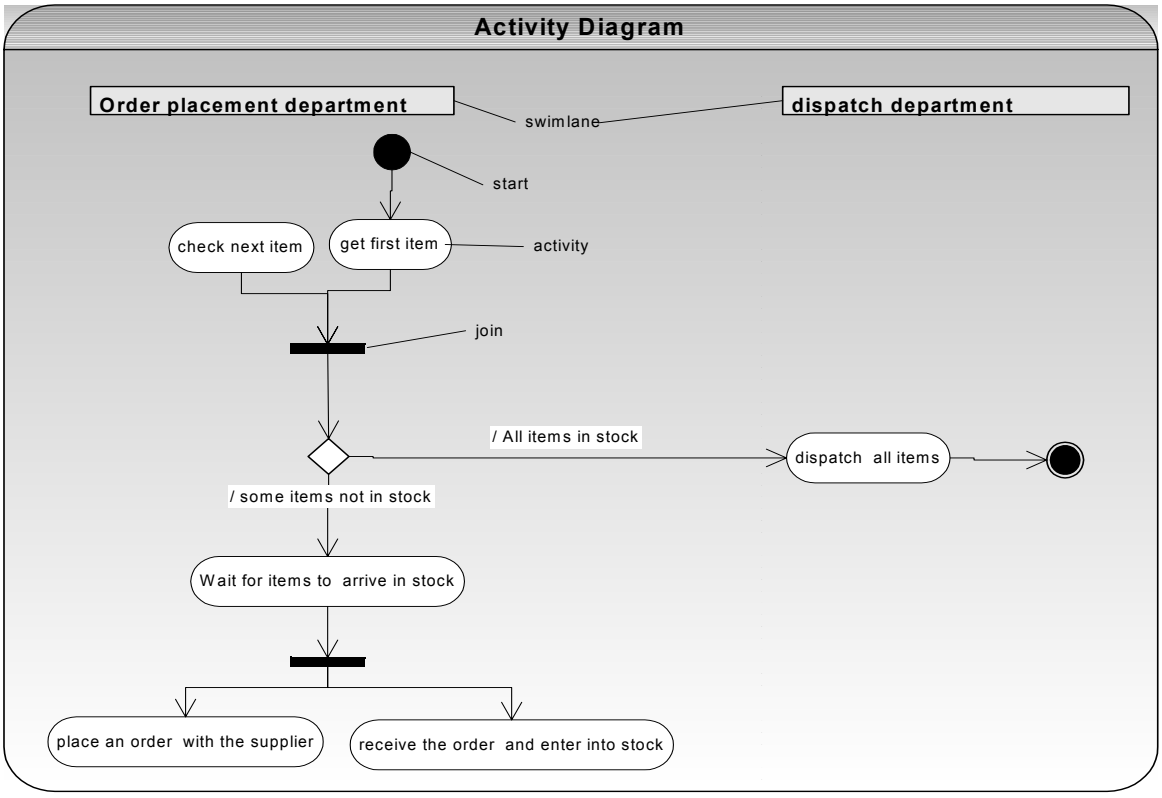


When to use statechart diagram?

- Statechart diagrams are good at describing the behaviour of an object across several use cases. But they are not good at describing the interaction or collaboration between many objects. Use interaction and/or activity diagrams in conjunction with a statechart diagram.

- Use it only for classes that have complex state changes and behaviour. **For example:** the User Interface (UI) control objects, Objects shared by multi-threaded programs etc.

Activity diagram: This is really a fancy flow chart. The activity diagram and statechart diagrams are related in a sense that statechart diagram focuses on object undergoing a transition process and an activity diagram focuses on the flow of activities involved in a single transition process.

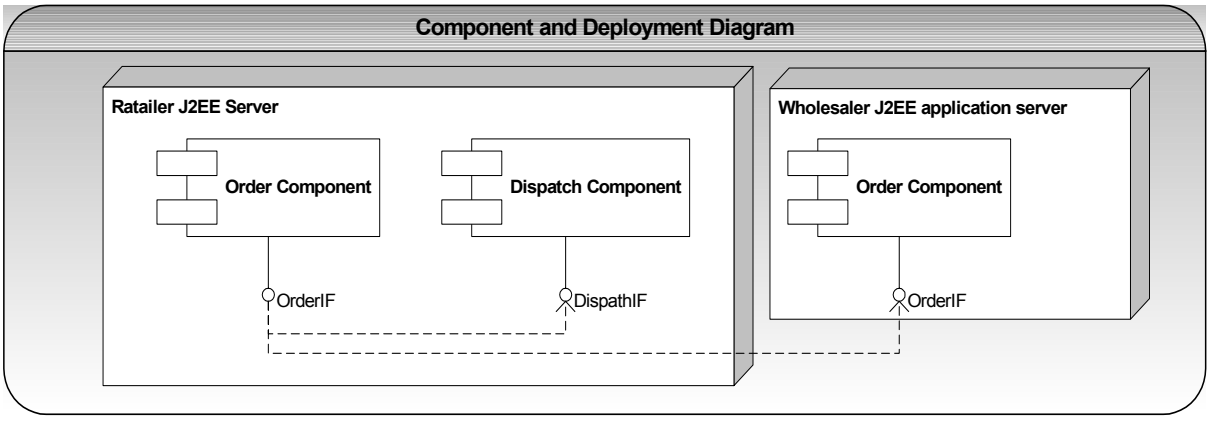


In domain modelling it is imperative that the diagram conveys which object (or class) is responsible for each activity. Activity diagrams can be divided into object **swimlanes** that determine which object is responsible for which activity. The swimlanes are quite useful because they combine the activity diagram's depiction of logic with the interaction diagram's depiction of responsibility. A single transition comes out of each **activity**, connecting to the next activity. A transition may **join** or **fork**.

When to use activity diagrams?

The **activity** and **statechart** diagrams are generally useful to express complex operations. The great strength of activity diagrams is that they support and encourage parallel behaviour. The activity and statechart diagrams are beneficial for workflow modelling with multi-threaded programming.

Component and **Deployment diagrams**: A component is a code module. Component diagrams are physical diagrams analogous to a class diagram. The deployment diagrams show the physical configuration of software and hardware components. The physical hardware is made up of **nodes**. Each **component** belongs to a node.



Q 108: What is the difference between aggregation and composition? SD DC

A 108:

Aggregation	Composition
Aggregation: An **association** in which one class belongs to another class or a collection. This is a **part of a whole** relationship where the **part** can exist without the **whole**. **For example:** A line item is whole and the products are the parts. If a line item is deleted then the products need not be deleted. (no cascade delete in database terms)	**Composition**: An **association** in which one class belongs to another class a collection. This is a **part of a whole** relationship where the **part** cannot exist without the **whole**. If the whole is deleted then the parts are deleted. **For example:** An Order is a whole and the line items are the parts. If an order is deleted then all the line items should be deleted as well (i.e. cascade deletes in database terms).
Aggregations are not allowed to be circular.	In a garbage-collected language like Java, The **whole** has the responsibility of **preventing** the garbage collector to prematurely collect the **part** by holding reference to it.

Q 109: What is the difference between a collaboration diagram and a sequence diagram? SD DC

A 109: You can automatically generate one from the other.

Sequence Diagram	Collaboration Diagram
The emphasis is on the **sequence**.	The emphasis is on the **object roles**

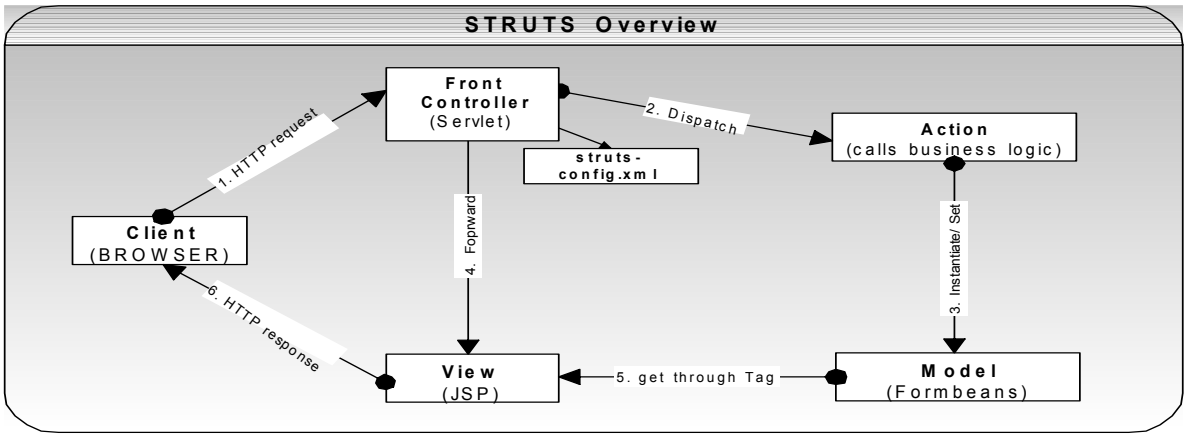
Reference: The above section on RUP & UML is based on the book **UML Distilled** by Martin Fowler and Kendall Scott. If you would like to have a good understanding of UML & RUP, then this book is recommended.

Enterprise - Struts

Struts is a Web-based user interface framework, which has been around for a few years. It is a matured and proven framework, which has been used in many J2EE projects. While Struts has been demonstrating its popularity, there is an emerging framework called **JavaServer Faces** (**JSF**) gaining lots of momentum and popularity. Like Struts, JSF provides Web application life cycle management through a controller servlet, and like Swing, JSF provides a rich component model complete with event handling and component rendering. So JSF can be considered as a combination of Struts frame work and Java Swing user interface framework. Refer **Q19 – Q20** in Emerging Technologies/Frameworks section for JSF.

Q 110: Give an overview of Struts? SF DP

A 110: Struts is a framework with set of cooperating classes, servlets and JSP tags that make up a reusable MVC 2 design.



- **Client (Browser):** A request from the client browser creates an HTTP request. The Web container will respond to the request with an HTTP response, which gets displayed on the browser.

- **Controller (ActionServlet class** and **Request Processor class):** The controller receives the request from the browser, and makes the decision where to send the request based on the **struts-config.xml**. **Design pattern:** Struts controller uses the **command design pattern** by calling the *Action* classes based on the configuration file struts-config.xml and the *RequestProcessor* class's process() method uses **template method design pattern** (Refer **Q11** in How would you go about … section) by calling a sequence of methods like:

  - **processPath(request, response)** → read the request URI to determine path element.
  - **processMapping(request,response)** → use the path information to get the action mapping
  - **processRoles(request,respose,mapping)** → Struts Web application security which provides an authorization scheme. By default calls request.isUserInRole(). For example allow /addCustomer action if the role is executive.

    ```
    <action path="/addCustomer" roles="executive">
    ```

  - **processValidate(request,response,form,mapping)** → calls the vaildate() method of the ActionForm.
  - **processActionCreate(request,response,mapping)**→ gets the name of the action class from the "type" attribute of the <action> element.
  - **processActionPerform(req,res,action,form,mapping)** → This method calls the execute method of the Action class which is where business logic is written.

- **Business Logic (Action class)**: The Servlet dispatches the request to Action classes, which act as a **thin wrapper to the business logic** (The actual business logic is carried out by either EJB session beans and/or plain Java classes). The action class helps control the **workflow** of the application. (**Note:** The Action class should only control the workflow and not the business logic of the application). The Action class uses the **Adapter design pattern** (Refer **Q11** in How would you go about … section).

- **ActionForm class**: Java representation of HTTP input data. They can carry data over from one request to another, but actually represent the data submitted with the request.

- **View (JSP):** The view is a JSP file. There is no business or flow logic and no state information. The JSP should just have tags to represent the data on the browser.

ActionServlet class is the controller part of the MVC implementation and is the core of the framework. It processes user requests, determines what the user is trying to achieve according to the request, pulls data from the model (if necessary) to be given to the appropriate view, and selects the proper view to respond to the user. As discussed above ActionServlet class delegates the grunt of the work to the **RequestProcessor** and **Action** classes.

The **ActionForm** class maintains the state for the Web application. ActionForm is an abstract class, which is subclassed for every input form model. The struts-config.xml file controls, which HTML form request maps to which ActionForm.

The **Action** class is a wrapper around the business logic. The purpose of the Action class is to translate the HttpServletRequest to the business logic. To use the Action class, subclass and overwrite the execute() method. The actual business logic should be in a separate package or EJB to allow reuse of business logic in protocol independent manner (ie the business logic should be used **not only** by HTTP clients **but also** by WAP clients, EJB clients, Applet clients etc).

The **ExceptionHandler** can be defined to execute when the Action class's execute() method throws an Exception. For example

```
<global-exceptions>
  <exception key="my.key" type="java.io.IOException" handler="my.ExceptionHandler"/>
</global-exceptions>
```

When an IOException is thrown then it will be handled by the execute() method of the my.ExceptionHandler class.

The struts-config.xml configuration information is translated into **ActionMapping**, which are put into the **ActionMappings** collection.
Further reading is recommended for more detailed understanding.

Q 111: What is a synchronizer token pattern in Struts or how will you protect your Web against multiple submissions? DC DP

A 111: Web designers often face the situation where a form submission must be protected against duplicate or **multiple submissions**. This situation typically occurs when the user clicks on submit button more than once before the response is sent back or client access a page by returning to the previously book marked page.

- The simplest solution that some sites use is that displaying a warning message "Wait for a response after submitting and do not submit twice.
- In the client only strategy, a flag is set on the first submission and from then onwards the submit button is disabled based on this flag. Useful in some situations but this strategy is coupled to the browser type and version etc.
- For a server-based solution the J2EE pattern **synchroniser token pattern** can be applied. The basic idea is to:

  1. Set a token in a session variable on the server side before sending the transactional page back to the client.
  2. The token is set on the page as a hidden field. On submission of the page first check for the presence of a valid token by comparing the request parameter in the hidden field to the token stored in the session. If the token is valid continue processing otherwise take other alternative action. After testing the token must be reset to null.

The synchroniser token pattern is implemented in Struts. How do we implement the alternate course of action when the second clicks on submit button will cancel the response from the first click. The thread for the first click still runs but has no means of sending the response back to the browser. This means the transaction might have gone through without notifying the user. The user might get the impression that transaction has not gone through.

Struts support for synchronisation comes in the form of:

ActionServlet.saveToken(HttpRequest**)** and **ActionServlet.isTokenValid(**HttpRequest**)** etc

---

Q 112: How do you upload a file in Struts? SF

A 112: In **JSP** page set the code as shown below: CO

```
<html:form action="upload.do" enctype="multipart/form-data" name="fileForm" type="FileForm"
        scope="session">
Please select the file that you would like to upload:
<html:file property="file" />
<html:submit />
</html:form>
```

In the **FormBean** set the code as shown below:

```
public class FileForm extends ActionForm {
   private FormFile file;

   public void setFile(FormFile file){
      this.file = file;
   }

   public FormFile getFile(){
     return file;
   }
}
```

Q 113: Are Struts action classes thread-safe? SF CI

A 113: No. Struts action classes are not thread-safe. Struts action classes are cached and reused for performance optimization at the cost of having to implement the action classes in a thread-safe manner.

Q 114: How do you implement internationalization in Struts? **SF**

A 114: Internationalization is built into Struts framework. In the JSP page set the code as shown below: **CO**

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<html:html locale="true">
<head>
  <title>i18n</title>
</head>

<body>
  <h2><bean:message key="page.title"/></h2>
</body>
</html:html>
```

Now we need to create an application resource file named **ApplicationResource.properties**.

```
page.title=Thank you for visiting!
```

Now in Italian, create an application resource file named **ApplicationResource_it.properties**.

```
page.title=Grazie per la vostra visita!
```

Finally, add reference to the appropriate resource file in the **struts-config.xml**.

Q 115: What is an action mapping in Struts? How will you extend Struts? **SF**

A 115: An **action mapping** is a configuration file (struts-config.xml) entry that, in general, associates an action name with an action. An action mapping can contain a reference to **a form bean** that the action can use, and can additionally define a list of local **forwards** that is visible only to this action.

How will you extend Struts?

Struts is not only a powerful framework but also very extensible. You can extend Struts in one or more of the following ways:

PlugIn: Define your own PlugIn class if you want to execute some init() and destroy() methods during the application startup and shutdown respectively. Some services like loading configuration files, initialising applications like logging, auditing, etc can be carried out in the init() method.

RequestProcessor: You can create your own RequestProcessor by extending the Struts RequestProcessor. For example you can override the processRoles(req, res, mapping) in your extended class if you want to query the LDAP server for the security authorization etc.

ActionServlet: You can extend the ActionServlet class if you want to execute your business logic at the application startup or shutdown or during individual request processing. You should take this approach only when the above mentioned approaches are not feasible.

Q 116: What design patterns are used in Struts? **DP**

A 116: Struts is based on model 2 MVC (Model-View-Controller) architecture. Struts controller uses the **command design pattern** (Refer **Q11** in How would you go about section) and the action classes use the **adapter design pattern**. The process() method of the RequestProcessor uses the **template method design pattern** (Refer **Q11** in How would you go about section). Struts also implement the following J2EE design patterns

- Service to Worker (Refer **Q25** in Enterprise section).
- Dispatcher View (Refer **Q25** in Enterprise section).
- Composite View (Struts Tiles) (Refer **Q25** in Enterprise section)
- Front Controller (Refer **Q24** in Enterprise section).
- View Helper (Refer **Q25** in Enterprise section).
- Synchronizer Token (Refer **Q111** in Enterprise section).

Enterprise - Web and Application servers

Q 117: What application servers, Web servers, LDAP servers, and Database servers have you used?
A 117:

Web Servers	Apache, Microsoft IIS, Netscape, Domino etc
Application Servers	IBM Websphere, BEA Weblogic, Apache Tomcat, Borland Enterprise Server, Fujitsu Interstage, JBoss, ATG Dynamo etc
LDAP Servers	IPlanet's directory server, SiemensDirX etc
Database Servers	IBM DB2, Oracle, SQL Server, Sybase, Informix

Q 118: What is the difference between a Web server and an application server? **SF**
A 118: In general, an application server prepares data for a Web server -- for example, gathering data from databases, applying relevant business rules, processing security checks, and/or storing the state of a user's session. The term application server may be misleading since the functionality isn't limited to applications. Its role is more as retriever and manager of data and processes used by anything running on a Web server. In the coming age of Web services, application servers will probably have an even more important role in managing service oriented components. One of the reasons for using an application server is to improve performance by off-loading tasks from a Web server. When heavy traffic has more users, more transactions, more data, and more security checks then more likely a Web server becomes a bottleneck.

Web Server	Application Server
Supports HTTP protocol. When a Web server receives an HTTP request, it responds with an HTTP response, such as sending back an HTML page (static content) or delegates the dynamic response generation to some other program such as CGI scripts or Servlets or JSPs in an application server.	Exposes **business logic** and **dynamic content** to a client through various protocols such as HTTP, TCP/IP, IIOP, JRMP etc.
Uses various scalability and fault-tolerance techniques.	Uses various scalability and fault-tolerance techniques. In addition provides resource pooling, component life cycle management, transaction management, messaging, security etc.

Q 119: What is a virtual host? **SF**
A 119: The term virtual host refers to the practice of maintaining **more than one server on one machine**. They are differentiated by their host names. You can have name based virtual hosts and IP address based virtual hosts. For example

A name-based "virtual host" has a **unique domain name**, but the same IP address. For example, www.company1.com and www.company2.com can have the same IP address 192.168.0.10 and share the same Web server. We can configure the Web server as follows:

```
NameVirtualHost 192.168.0.10

<VirtualHost 192.168.0.10>
  ServerName www.company1.com
  DocumentRoot  /web/company1
</VirtualHost>

<VirtualHost 192.168.0.10>
  ServerName www.company2.com
  DocumentRoot  /web/company2
</VirtualHost>
```

In this scenario, both www.company1.com and www.company2.com are registered with the standard **domain name service** (**DNS**) registry as having the IP address 192.168.0.10. A user types in the URL http://www.company1.com/hello.jsp in their browser. The user's computer resolves the name

www.company1.com to the IP address 192.168.0.10. The Web server on the machine that has the IP address 192.168.0.10, so it receives the request. The Web server determines which virtual host to use by matching the request URL It gets from an HTTP header submitted by the browser with the “ServerName” parameter in the configuration file shown above.

Name-based virtual hosting is usually easier, since you have to only configure your DNS server to map each hostname to a single IP address and then configure the Web server to recognize the different hostnames as discussed in the previous paragraph. Name-based virtual hosting also eases the demand for scarce IP addresses limited by physical network connections [but modern operation systems supports use of virtual interfaces, which are also known as IP aliases]. Therefore you should use name-based virtual hosting unless there is a specific reason to choose IP-based virtual hosting. Some reasons why you might consider using IP-based virtual hosting:

- Name-based virtual hosting cannot be used with SSL based secure servers because of the nature of the SSL protocol.
- Some operating systems and network equipment implement bandwidth management techniques that cannot differentiate between hosts unless they are on separate IP addresses.
- IP based virtual hosts are useful, when you want to manage more than one site (like live, demo, staging etc) on the same server where hosts inherit the characteristics defined by your main host. But when using SSL for example, a unique IP address is necessary.

For example in development environment when using the test client and the server on the same machine we can define the host file as shown below:

UNIX user: /etc/hosts
Windows user: C:\WINDOWS\SYSTEM32\DRIVERS\ETC\HOSTS

```
127.0.0.1       localhost
127.0.0.1       www.company1.com
127.0.0.1       www.company2.com
```

[Reference: http://httpd.apache.org/docs/1.3/vhosts/]

Q 120: What is application server clustering? **SI**

A 120: An application server cluster consists of a number of application servers loosely coupled on a network. The server cluster or server group is generally distributed over a number of machines or nodes. The important point to note is that the cluster appears as a single server to its clients.

The goals of application server clustering are:

- **Scalability:** should be able to add new servers on the existing node or add new additional nodes to enable the server to handle increasing loads without performance degradation, and in a manner transparent to the end users.
- **Load balancing:** Each server in the cluster should process a fair share of client load, in proportion to its processing power, to avoid overloading of some and under utilization of other server resources. Load distribution should remain balanced even as load changes with time.
- **High availability:** Clients should be able to access the server at almost all times. Server usage should be transparent to hardware and software failures. If a server or node fails, its workload should be moved over to other servers, automatically as fast as possible and the application should continue to run uninterrupted. This method provides a fair degree of application system fault-tolerance. After failure, the entire load should be redistributed equally among working servers of the system.

[Good read: Uncover the hood of J2EE clustering by Wang Yu on http://www.theserverside.com]

Q 121: Explain Java Management Extensions (**JMX**)? **SF**

A 121: JMX framework can improve the manageability of your application by

- Monitoring your application for performance problems, critical events, error condition statistics, etc. **For example** you can be notified if there is a sudden increase in traffic or sudden drop in performance of your website.
- Making your application more controllable and configurable at runtime by directly exposing application API and parameters. **For example** you could switch your database connection to an alternate server. You can also change the level of debugging and logging within the application without stopping the server.
- By interfacing JMX to your hardware, database server and application server, health checks can be performed of your infrastructure.

Q 122: Explain some of the portability issues between different application servers? **SI**
A 122: Transaction isolation levels, lazy loading and dirty marker strategies for EJB, class loading visibility etc.

Enterprise - Best practices and performance considerations

Q 123: Give some tips on J2EE application server performance tuning? **PI**
A 123:

- Set the Web container threads, which will be used to process incoming HTTP requests. The minimum size should be tuned to handle the average load of the container and maximum should be tuned to handle the peak load. The maximum size should be less than or equal to the number of threads in your Web server.
- When an EJB is called from a servlet or another EJB within the same JVM (i.e. same application server) then performance can be improved by running EJBs in pass-by-reference mode as oppose to pass-by-value which is the default mode. Care should be taken to test the application properly before going into production because some valid applications may not work correctly when pass-by-reference setting is switched on.
- Application servers maintain a pool of JDBC resources so that a new connection does not need to be created for each transaction. Application servers can also cache your prepared statements to improve performance. So you can tune the minimum and maximum size of these pools.
- Tune your initial heap size for the JVM so that the garbage collector runs at a suitable interval so that it does not cause any unnecessary overhead. Adjust the value as required to improve performance.
- Set the session manager settings appropriately based on following guidelines:
  - Set the appropriate value for in memory session count.
  - Reduce the session size.
  - Don't enable session persistence unless required by your application.
  - Invalidate your sessions when you are finished with them by setting appropriate session timeout.
- Calls to EJB from a separate JVM are handled by ORB (Object Request Broker). ORB uses a pool of threads to handle these requests. The thread pool size should be set appropriately to handle average and peak loads.
- If a servlet or JSP file is called frequently with identical URL parameters then they can be dynamically cached to improve performance.
- Turn the application server tracing off unless required for debugging.
- Some application servers support lazy loading and dirty marker strategies with EJB to improve performance.

Q 124: Explain some of the J2EE best practices? **BP**
A 124:

- **Recycle your valuable resources by either pooling** or **caching**. You should create a limited number of resources and share them from a common pool (e.g. pool of threads, pool of database connections, pool of objects etc). Caching is simply another type of pooling where instead of pooling a connection or object, you are pooling remote data (database data) and placing it in the memory (using Hashtable etc).

- **Avoid embedding business logic in a protocol dependent manner** like in JSPs, HttpServlets, Struts action classes etc. This is because your business logic should be not only executed by your Web clients but also required to be shared by various GUI clients like Swing based stand alone application, WAP clients etc.

- **Automate** the build process with tools like **Ant, CruiseControl,** and **Maven** etc. In an enterprise application the build process can become quite complex and confusing.

- **Build** test cases first (i.e. Test Driven Development (TDD), refer section Emerging Technologies) using tools like **JUnit**. Automate the testing process and integrate it with build process.

- **Separate HTML code from the Java code**: Combining HTML and Java code in the same source code can make the code less readable. Mixing HTML and scriplet will make the code extremely difficult to read and maintain. The display or behaviour logic can be implemented as a custom tags by the Java developers and Web designers can use these Tags as the ordinary XHTML tags.

- It is best practice to use multi-threading and stay away from **single threaded model of the servlet** unless otherwise there is a compelling reason for it. Shared resources can be synchronized or used in read-only manner or shared values can be stored in a database table. Single threaded model can adversely affect performance.

- **Apply the following JSP best practices**:

  - **Place data access logic in JavaBeans**: The code within the JavaBean is readily accessible to other JSPs and Servlets.

  - **Factor shared behaviour out of Custom Tags into common JavaBeans classes**: The custom tags are not used outside JSPs. To avoid duplication of behaviour or business logic, move the logic into JavaBeans and get the custom tags to utilize the beans.

  - **Choose the right** "***include"*** **mechanism:** What are the differences between static and a dynamic include? Using includes will improve code reuse and maintenance through modular design. Which one to use? Refer **Q31** in Enterprise section.

  - **Use style sheets** (e.g. css)**, template mechanism** (e.g. struts tiles etc) and **appropriate comments** (both hidden and output comments)**.**

- If you are using EJBs apply the EJB best practices as described in **Q82** in Enterprise section.

- Use the J2EE standard packaging specification to improve portability across Application Servers.

- Use proven frameworks like **Struts, Spring, Hibernate, JSF** etc.

- Apply appropriate proven J2EE design patterns to improve performance and minimise network communications cost (Session façade pattern, Value Object pattern etc).

- Batch database requests to improve performance. For example

```
Connection con = DriverManager.getConnection(……).
Statement stmt = con.createStatement().
stmt.addBatch("INSERT INTO  Address……");
stmt.addBatch("INSERT INTO  Contact……");
stmt.addBatch("INSERT INTO  Personal");
int[] countUpdates =  stmt.executeBatch();
```

  Use "**PreparedStatements"** instead of ordinary "Statements" for repeated reads.

- Avoid resource leaks by

  - Closing all database connections after you have used them.
  - Clean up objects after you have finished with them especially when an object having a long life cycle refers to a number of objects with short life cycles (you have the potential for memory leak).

- Poor exception handling where the connections do not get closed properly and clean up code that never gets called. You should put clean up code in a **finally** {} block.

- Handle and propagate exceptions correctly. Decide between checked and unchecked (i.e RunTime exceptions) exceptions.

Q 125: Explain some of the J2EE best practices to improve performance? **BP** **PI**

A 125: In short manage valuable resources wisely and recycle them where possible, minimise network overheads and serialization cost, and optimise all your database operations.

- **Manage and recycle your valuable resources by either pooling or caching**. You should create a limited number of resources and share them from a common pool (e.g. pool of threads, pool of database connections, pool of objects etc). Caching is simply another type of pooling where instead of pooling a connection or object, you are pooling remote data (database data), and placing it in memory (using Hashtable etc). Unused stateful session beans must be removed explicitly and appropriate idle timeout should be set to control stateful session bean life cycle.

- **Use effective design patterns to minimise network overheads** (Session facade, Value Object etc Refer **Q84, Q85** in Enterprise section), use of fast-lane reader pattern for database access (Refer **Q86** in Enterprise section). Caching of retrieved JNDI InitialContexts, factory objects (e.g. EJB homes) etc. using the service locator design pattern, which reduces expensive JNDI access with the help of caching strategies.

- **Minimise serialization costs** by marking references (like file handles, database connections etc), which do not required serialization by declaring them '**transient**' (Refer **Q19** in Java section). Use pass-by-reference where possible as opposed to pass by value.

- **Set appropriate timeouts:** for the HttpSession objects, after which the session expires, set idle timeout for stateful session beans etc.

- Improve the **performance of database operations** with the following tips:

  - Database connections should be released when not needed anymore, otherwise there will be potential resource leakage problems.

  - Apply least restrictive but valid transaction isolation level.

  - Use JDBC prepared statements for overall database efficiency and for batching repetitive inserts and updates. Also batch database requests to improve performance.

  - When you first establish a connection with a database by default it is in auto-commit mode. For better performance turn auto-commit off by calling the connection.setAutoCommit(false) method.

  - Where appropriate (you are loading 100 objects into memory but use only 5 objects) **lazy load** your data to avoid loading the whole database into memory using the virtual proxy pattern. Virtual proxy is an object, which looks like an object but actually contain no fields until when one of its methods is called does it load the correct object from the database.

  - Where appropriate **eager load** your data to avoid frequently accessing the database every time over the network.

Enterprise – Logging, testing and deployment

Q 126: Give an overview of log4J? **SF**

A 126: Log4j is a logging framework for Java. Log4J is designed to be fast and flexible. Log4J has 3 main components which work together to enable developers to log messages:

- Loggers [was called *Category* prior to version 1.2]
- Appenders
- Layout

Logger: The foremost advantage of any logging API like log4J, apache commons logging etc over plain System.out.println is its ability to disable certain log statements while allowing others to print unhindered. Loggers are hierarchical. The root logger exists at the top of the hierarchy. The root logger always exists and it cannot be retrieved by name. The hierarchical nature of the logger is denoted by "." notation. For example the logger "java.util" is the parent of child logger "java.util.Vector" and so on. Loggers may be assigned priorities such as DEBUG, INFO, WARN, ERROR and FATAL. If a given logger is not assigned a priority, then it inherits the priority from its closest ancestor. The logging requests are made by invoking one of the following printing methods of the logger instance: debug(), info(), warn(), error(), fatal().

Appenders and Layouts: In addition to selectively enabling and disabling logging requests based on the logger, the log4J allows logging requests to multiple destinations. In log4J terms the output destination is an appender. There are appenders for console, files, remote sockets, JMS, etc. One logger can have more than one appender. A logging request for a given logger will be forwarded to all the appenders in that logger plus the other appenders higher in the hierarchy. In addition to the output destination the output format can be categorised as well. This is accomplished by associating layout with an appender. The layout is responsible for formatting the logging request according to user's settings.

Sample configuration file:

```
#set the root logger priority to DEBUG and its appender to App1
log4j.rootLogger=DEBUG, App1

#App1 is set to a console appender
log4j.appender.App1=org.apache.log4j.ConsoleAppender

#appender App1 uses a pattern layout
log4j.appender.App1.layout=org.apache.log4j.PatternLayout.
log4j.appender.App1.layout.ConversionPattern=%-4r [%t] %-5p %c %x -%m%n

# Print only messages of priority WARN or above in the package com.myapp
log4j.Logger.com.myapp=WARN
```

XML configuration for log4j is available, and is usually the best practise.

Q 127: How do you initialize and use Log4J? SF CO

A 127:

```
public class MyApp {
   //Logger is a utility wrapper class to be written with appropriate printing methods
   static Logger log = Logger.getLogger (MyApp.class.getName());

   public void my method() {
        if(log.isDebugEnabled())
              log.debug("This line is reached……..." + var1 + "-" + var2);
        )
   }

}
```

Q 128: What is the hidden cost of parameter construction when using Log4J? SF PI

A 128:

Do not use in frequently accessed methods or loops: CO

```
log.debug ("Line number" + intVal +  " is less than " +  String.valueOf(array[i]));
```

The above construction has a **performance cost in frequently accessed methods and loops** in constructing the message parameter, concatenating the *String* etc regardless of whether the message will be logged or not.

Do use in frequently accessed methods or loops: CO

```
If (log.isDebugEnabled()) {
     log.debug ("Line number" + intVal +  " is less than " +  String.valueOf(array[i]));
}
```

The above construction will avoid the parameter construction cost by only constructing the message parameter when you are in debug mode. But it is not a best practise to place log.isDebugEnabled() around all debug code.

Q 129: What is the test phases and cycles? **SD**

A 129:

- **Unit tests** (e.g. JUnit etc, carried out by developers).
  There are two popular approaches to testing server-side classes: **mock objects**, which test classes by simulating the server container, and **in-container** testing, which tests classes running in the actual server container. If you are using Struts framework, *StrutsTestCase* for JUnit allows you to use either approach, with very minimal impact on your actual unit test code.

- **System tests or functional tests** (carried out by business analysts and/or testers).
- **Integration tests** (carried out by business analysts, testers, developers etc).
- **Regression tests** (carried out by business analysts and testers).
- **Stress volume tests or load tests** (carried out by technical staff).
- **User acceptance tests** (UAT – carried out by end users).

Each of the above test phases will be carried out in cycles. Refer **Q13** in How would you go about… section for JUnit, which is an open source unit-testing framework.

Q 130: Brief on deployment environments you are familiar with?

A 130: Differ from project team to project team [**Hint**] :

Application environments where "ear" files get deployed.

Development box: can have the following instances of environments in the same machine (need not be clustered).

- Development environment → used by developers.
- System testing environment → used by business analysts.

Staging box: can have the following instances of environments in the same machine (preferably clustered servers with load balancing)

- Integration testing environment → used for integration testing, user acceptance testing etc.
- Pre-prod environment → used for user acceptance testing, regression testing, and load testing or stress volume testing (SVT). [This environment should be exactly same as the production environment].

Production box:

- Production environment → live site used by actual users.

Data environments (Database)

Note: Separate boxes [not the same boxes as where applications (i.e. ear files) are deployed]

- **Development box** (database).
  Used by applications on development and system testing environments. Separate instances can be created on the same box for separate environments like development and system testing.

- **Staging Box** (database)
  Used by applications on integration testing and user acceptance testing environments. Separate instances can be created on the same box for separate environments.

- **Production Box** (database)
  Live data used by actual users of the system.

Enterprise - Personal

Q 131: Tell me about yourself or about some of the recent projects you have worked with? What do you consider your most significant achievement? Why do you think you are qualified for this position? Why should we hire you and what kind of contributions will you make?

A 131: **[Hint:]** Pick your recent projects and give a brief overview of it. Also it is imperative that during your briefing that you demonstrate how you applied your skills and knowledge in some of the following key areas and fixed any issues.

- **Design Concepts**: Refer **Q02, Q03, Q19, Q20, Q21, Q91, Q98,** and **Q101**.
- **Design Patterns**: Refer **Q03, Q24, Q25, Q83, Q84, Q85, Q86, Q87, Q88** and **Q111**.
- **Performance issues**: Refer **Q10, Q16, Q45, Q46, Q97, Q98, Q100, Q123**, and **Q125**.
- **Memory issues**: Refer **Q45** and **Q93**
- **Multi-threading (Concurrency issues)**: Refer **Q16, Q34,** and **Q113**
- **Exception Handling**: Refer **Q76** and **Q77**
- **Transactional issues**: Refer **Q43, Q71, Q72, Q73, Q74, Q75** and **Q77**.
- **Security issues:** Refer **Q23, Q58,** and **Q81**
- **Scalability issues:** Refer **Q20, Q21, Q120** and **Q122**.
- **Best practices:** Refer **Q10, Q16, Q39, Q40, Q46, Q82, Q124,** and **Q125**

Refer **Q66 – Q72** in Java section for frequently asked non-technical questions.

Q 132: Have you used any load testing tools?

A 132: Rational Robot, JMeter, LoadRunner, etc.

Q 133: What source control systems have you used? **SD**

A 133: CVS, VSS (Visual Source Safe), Rational clear case etc. Refer **Q13** in How would you go about section…. for CVS.

Q 134: What operating systems are you comfortable with? **SD**

A 134: NT, Unix, Linux, Solaris etc

Q 135: Which on-line technical resources do you use to resolve any design and/or development issues?

A 135: http://www.theserverside.com, http://www.javaworld.com, http://www-136.ibm.com/developerworks/Java/, http://java.sun.com/, www.javaperformancetuning.com etc

---

Enterprise – Software development process

Q 136: What software development processes/principles are you familiar with? **SD**

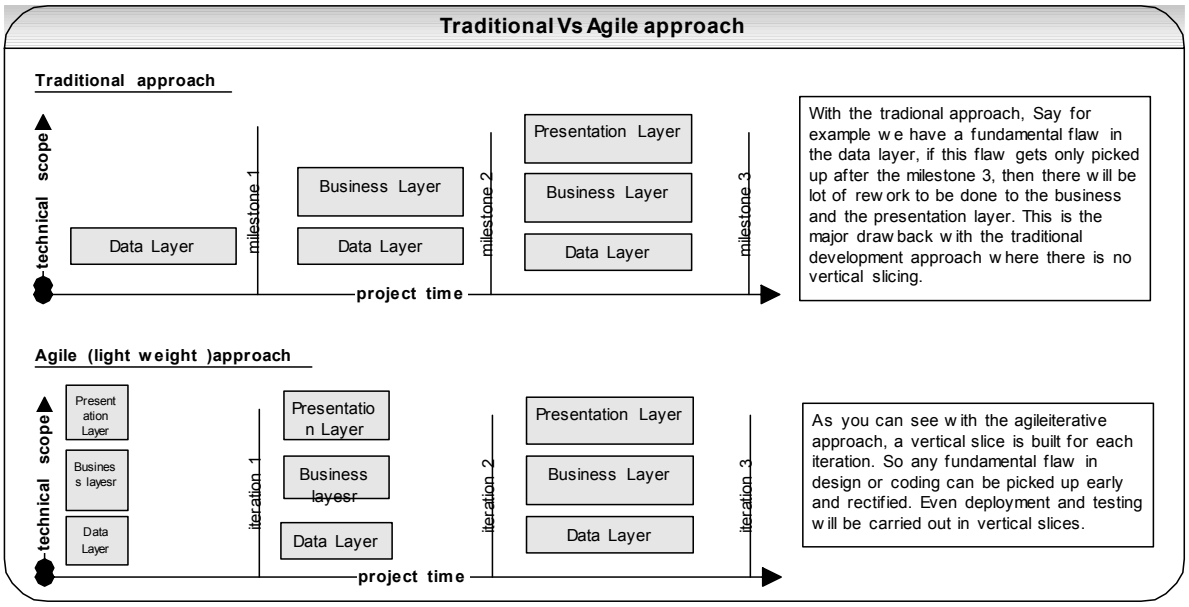
A 136: **Agile (i.e. lightweight) software development process** is gaining popularity and momentum across organizations.

Agile software development manifesto → [Good read: http://www.agilemanifesto.org/principles.html].

- Highest priority is to satisfy the customer.
- Welcome requirement changes even late in development life cycle.
- Business people and developers should work collaboratively.
- Form teams with motivated individuals who produce best designs and architectures.

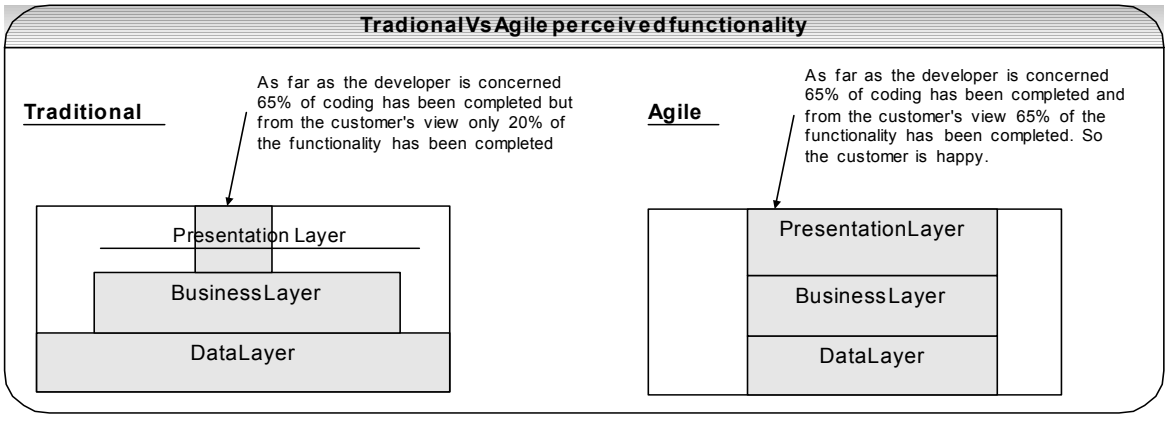
- Teams should be pro-active on how to become more effective without becoming complacent.
- Quality working software is the primary measure of progress.

Why is iterative development with vertical slicing used in agile development? Your overall software quality can be improved through iterative development, which provides you with constant feedback.



Several methodologies fit under this agile development methodology banner. All these methodologies share many characteristics like **iterative and incremental development**, **test driven development**, **stand up meetings to improve communication**, **automatic testing, build and continuous integration of code** etc. Among all the agile methodologies XP is the one which has got the most attention. Different companies use different flavours of agile methodologies by using different combinations of methodologies.

How does vertical slicing influence customer perception? With the iterative and incremental approach, customer will be comfortable with the progress of the development as opposed to traditional big bang approach.



- E**X**treme **P**rogramming [**XP**] → simple design, pair programming, unit testing, refactoring, collective code ownership, coding standards, etc. Refer **Q10** in "How would you go about…" section. XP has four key values: Communication, Feedback, Simplicity and Courage. It then builds up some tried and tested practices and techniques. XP has a strong emphasis on testing where tests are integrated into continuous integration and build process, which yields a highly stable platform. XP is designed for smaller teams of 20 – 30 people.

- **RUP** (Rational Unified Process) → Model driven architecture, design and development; customizable frameworks for scalable process; iterative development methodology; Re-use of architecture, code, component, framework, patterns etc. RUP can be used as an agile process for smaller teams of 20-30

people, or as a heavy weight process for larger teams of 50-100 people. Refer **Q103 – Q105** in Enterprise section.

- **F**eature **D**riven **D**evelopment [**FDD**] → Jeff De Luca and long time OO guru Peter Coad developed feature Driven Development (FDD). Like the other **adaptive methodologies**, it focuses on short iterations that deliver tangible functionality. FDD was originally designed for larger project teams of around 50 people. In FDD's case the iterations are two weeks long. FDD has five processes. The first three are done at the beginning of the project. The last two are done within each iteration.

    1. Develop an Overall Model
    2. Build a Features List
    3. Plan by Feature
    4. Design by Feature
    5. Build by Feature

  The developers come in two kinds: class owners and chief programmers. The chief programmers are the most experienced developers. They are assigned features to be built. However they don't build them alone. Instead the chief programmer identifies which classes are involved in implementing the feature and gathers their class owners together to form a feature team for developing that feature. The chief programmer acts as the coordinator, lead designer, and mentor while the class owners do much of the coding of the feature.

- Test Driven Development [**TDD**] → TDD is an iterative software development process where you **first write the test with the idea that it must fail**. Refer **Q1** in Emerging Technologies/Frameworks section…

- **Scrum** → Scrum divides a project into sprints (aka iterations) of 30 days. Before you begin a sprint you define the functionality required for that sprint and leave the team to deliver it. But every day the team holds a short (10 – 15 minute) meeting, called a scrum where the team runs through what it will achieve in the next day. Some of the questions asked in the scrum meetings are:

    - What did you do since the last scrum meetings?
    - Do you have any obstacles?
    - What will you do before next meeting?

  This is very similar to stand-up meetings in XP and iterative development process in RUP.

Enterprise – Key Points

- J2EE is a **3-tier** (or **n-tier**) system. Each tier is logically separated and loosely coupled from each other, and may be distributed.

- J2EE applications are developed using **MVC architecture**, which divides the functionality of displaying and maintaining of the data to minimise the degree of coupling between enterprise components.

- J2EE modules are deployed as ear, war and jar files, which are standard application deployment archive files.

- HTTP is a stateless protocol and state can be maintained between client requests using HttpSession, URL rewriting, hidden fields and cookies. HttpSession is the recommended approach.

- Servlets and JSPs are by default multi-threaded, and care should be taken in declaring instance variables and accessing shared resources. It is possible to have a single threaded model of a servlet or a JSP but this can adversely affect performance.

- Clustering promotes high availability and scalability. The considerations for servlet clustering are:

    - Objects stored in the session should be serializable.
    - Design for idempotence.
    - Avoid using instance and static variables in read and write mode.
    - Avoid storing values in the *ServletContext*.
    - Avoid using java.io.* and use getResourceAsStream() instead.

- JSPs have a translation or a compilation process where the JSP engine translates and compiles a JSP file into a JSP servlet.

- JSPs have 4 different **scope** values: page, request, session and application. JSPs can be included **statically**, where all the included JSP pages are compiled into a single servlet during the translation or compilation phase or included **dynamically,** where included JSPs are compiled into separate servlets and the content generated by these servlets are included at runtime in the JSP response.

- Avoid scriptlet code in your JSPs and use **JavaBeans** or **custom tags** (e.g. Struts tags, JSTL tags, JSF tags etc) instead.

- Databases can run out cursors if the connections are not closed properly. The valuable resources like connections and statements should be enclosed in a **try{}** and **finally{}** block.

- Prepared statements offer better performance as opposed to statements, as they are **precompiled** and **reuse the same execution plan** with different arguments. Prepared statements are also more secure because they use bind variables, which can prevent SQL injection attacks.

- JNDI provides a generic interface to LDAP and other directory services like NDS, DNS etc.

- In your code always make use of a **logical JNDI reference** (java:comp/env/ejb/MyBean) as opposed to **physical JNDI reference** (ejb/MyBean) because you cannot guarantee that the physical JNDI location you specify in your code will be available. Your code will break if the physical location is changed.

- LDAP servers are typically used in J2EE applications to authenticate and authorise users. LDAP servers are hierarchical and are **optimized for read access**, so likely to be faster than database in providing read access.

- RMI facilitates object method calls between JVMs. JVMs can be located on separate host machines, still one JVM can invoke methods belonging to an object residing in another JVM (i.e. address space). RMI uses object serialization to marshal and unmarshal parameters. The remote objects should extend the *UnicastRemoteObject*.

- To go through a firewall, the RMI protocol can be embedded within the firewall trusted HTTP protocol, which is called **HTTP tunnelling**.

- EJB (i.e. 2.x) is a remote, distributed multi-tier system, which supports protocols like JRMP, IIOP, and HTTP etc. EJB components contain business logic and system level supports like security, transaction, instance pooling, multi-threading, object life-cycles etc are managed by the EJB container and hence simplify the programming effort. Having said this, there are emerging technologies like:

    - Hibernate, which is an open source object-to-relational (O/R) mapping framework.
    - EJB 3.0, which is taking ease of development very seriously and has adjusted its model to offer the plain old Java objects (i.e. POJOs) based persistence and the new O/R mapping model based on hibernate.

  Refer **Q14 – Q18** in Emerging technologies / Frameworks section for brief discussion on hibernate and EJB 3.0.

- EJB transaction attributes (like Required, Mandatory, RequiresNew, Supports etc) are specified declaratively through EJB deployment descriptors. Isolation levels are not part of the EJB 2.x specification. So the isolation levels can be set on the resource manager either explicitly on the *Connection* or via the application server specific configuration.

- A transaction is often described by **ACID** (Atomic, Consistent, Isolated and Durable) properties. A **distributed transaction** is an ACID transaction between two or more independent transactional resources like two separate databases. A **2-phase commit** is an approach for committing a distributed transaction in 2 phases.

- EJB 2.x has two types of exceptions:

    - **System exception**: is an unchecked exception derived from *java.lang.RuntimeException*. It is thrown by the system and is not recoverable.
    - **Application exception**: is specific to an application and is thrown because of violation of business rules.

- EJB container managed transactions are automatically rolled back when a system exception occurs. This is possible because the container can intercept system exceptions. However when an application exception occurs, the container does not intercept and leaves it to the code to roll back using ctx.setRollbackOnly() method.

- EJB containers can make use of lazy **loading** (i.e. not creating an object until it is accessed) and **dirty marker** (ie persist only the entity beans that have bean modified) strategies to improve entity beans performance.

- Message Oriented Middleware (MOM) is a software infrastructure that asynchronously communicates with other disparate systems through the production and consumption of messages. Messaging enables loosely coupled distributed communication. Java Messaging Service (**JMS**) is a Java API that allows applications to create, send, receive read messages in a standard way, hence improves portability.

- Some of the design decisions you need to make in JMS are message acknowledgement modes, transaction modes, delivery modes etc, synchronous vs. asynchronous paradigm, message body types, setting appropriate timeouts etc.

- XML documents can be processed in your Java/J2EE application either using a SAX parser, which is event driven or a DOM parser, which creates a tree structure in memory. The other XML related technologies are DTD, XSD, XSL, XPath, etc and Java and XML based technologies are JAXP, JAXB etc.

- There is an impedance mismatch between object and relational technology. Classes represent both data and behaviour whereas relational database tables just implement data. Inheritance class structure can be mapped to relational data model in one of the following ways:

  - Map class hierarchy to single database table.
  - Map each class to its own table.
  - Map each concrete class to its own table
  - Generic meta-data driven approach.

- Normalize data in your database for accuracy and denormalize data in your database for performance.

- **RUP** (Rational Unified Process) has 4 phases in the following order Inception, Elaboration, Construction, and Transition. Agile (i.e. lightweight) software development process is gaining popularity and momentum across organizations. Several methodologies like XP, RUP, Scrum, FDD, TDD etc fit under this agile development methodology banner. All these methodologies share many characteristics like iterative and incremental development, stand-up meetings to improve communication, automatic build, testing and continuous integration etc.

- UML is applicable to the object oriented (OO) problem solving. There are different types of UML diagrams like use case diagrams, class diagrams, sequence diagrams, collaboration diagrams, state chart diagrams, activity diagrams, component diagrams, deployment diagrams etc.

- Class diagrams are vital within OO methods. Class diagrams have the following possible relationships, association, aggregation, composition, generalization, and dependency.

- Struts is an MVC framework. Struts action classes are not thread-safe and care should be taken in declaring instance variables or accessing other shared resources. JSF is another Web UI framework like Struts gaining popularity and momentum.

- Log4j has three main components: *loggers*, *appenders* and *layouts*. Logger is a utility wrapper class. JUnit is an open source unit-testing framework.

- You can improve the performance of a J2EE application as follows :

  1. Manage and recycle your valuable resources like connections, threads etc by either pooling or caching.
  2. Use effective design patterns like session façade, value object, fast lane reader etc to minimise network overheads.
  3. Set appropriate timeouts for HttpSession objects.
  4. Use JDBC prepared statements as opposed to statements.
  5. Release database connections in a finally {} block when finished.
  6. Apply least restrictive but valid transaction isolation level.
  7. Batch database requests.
  8. Minimise serialization costs by marking references like file handles, database connections, etc which do not require serialization by declaring them transient.

- Some of the J2EE best practices are:

  1. Recycle your valuable resources by either pooling or caching.
  2. Automate your build process with tools like Ant, CruiseControl, and Maven etc, and continuously integrate your code into your build process.
  3. Build test cases first using tools like JUnit.
  4. Use standard J2EE packaging to improve portability.
  5. Apply appropriate proven design patterns.

6. Use proven frameworks like Struts, Spring, Hibernate, JSF, JUnit, Log4J, etc.
7. Handle and propagate exceptions correctly.
8. Avoid resource leaks by closing all database connections after you have used them.

- The goals of application server clustering are to achieve scalability, load balancing, and high availability.

- Java Management Extension (JMX) framework can improve the manageability of your application, for performance problems, critical events, error conditions etc and perform health checks on your hardware, database server etc. You can also configure and control your application at runtime.

- Finally get familiarised with some of the key Java & J2EE **design patterns** like:

1. **MVC design pattern**: J2EE uses this design pattern or architecture.

2. **Chain of responsibility design pattern**: Servlet filters use a slightly modified version of chain of responsibility design pattern.

3. **Front controller J2EE design pattern**: provides a centralized access point for HTTP request handling to support the integration system services like security, data validation etc. This is a popular J2EE design pattern.

4. **Composite view J2EE design pattern**: creates an aggregate view from atomic sub-views.

5. **View helper J2EE design pattern**: avoids duplication of code. The helper classes are JavaBeans and custom tags (e.g. Struts tags, JSF tags, JSTL tags etc).

6. **Service to worker and dispatcher view J2EE design pattern**: These two patterns are a combination of front controller and view helper patterns with a dispatcher component. These two patterns differ in the way they suggest different division of responsibility among components.

7. **Bridge design pattern**: Java Data Base Connectivity (JDBC) uses the bridge design pattern. The JDBC API provides an abstraction and the JDBC drivers provide the implementation.

8. **Proxy design pattern**: RMI & EJB uses the proxy design pattern. A popular design pattern.

9. **Business delegate J2EE design pattern**: used to reduce the coupling between the presentation tier and the business services tier components.

10. **Session façade J2EE design pattern**: too many fine-grained method calls between the client and the server will lead to network overhead and tight coupling. Use a session bean as a façade to provide a coarse-grained service access layer to clients.

11. **Value object J2EE design pattern**: avoid fine-grained method calls by creating a value object, which will help the client, make a coarse-grained call.

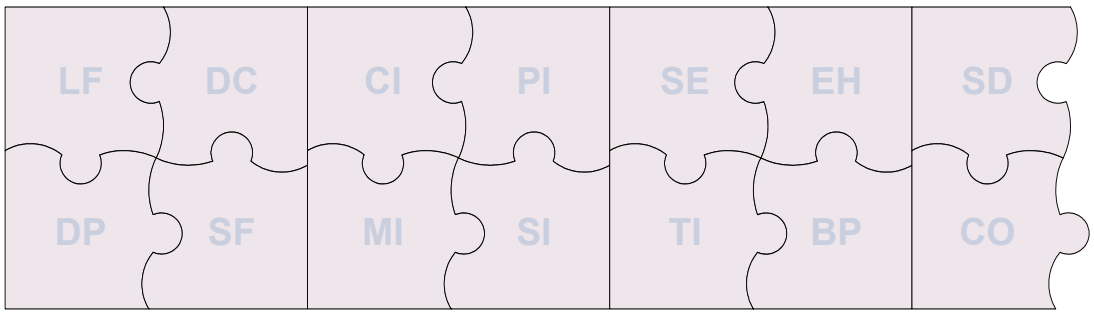
12. **Fast-lane reader J2EE design pattern**: access the persistence layer directly using a DAO (Data Access Object) pattern instead of using entity beans.

13. **Service locator J2EE design pattern**: expensive and redundant JNDI lookups can be avoided by caching and reusing the already looked up service objects.

Recommended reading on J2EE design patterns:

- Core J2EE Patterns: Best Practices and Design Strategies, Second Edition (Hardcover) by Deepak Alur, Dan Malks, John Crupi.

Let us put all together in the next section



SECTION THREE

How would you go about…?

- This section basically assesses your knowledge of how to perform certain tasks like documenting your project, identifying any potential performance, memory, transactional, and/or design issues etc.
- It also assesses if you have performed any of these tasks before. If you have not done a particular task, you can demonstrate that you know how to go about it if the task is assigned to you.
- This section also recaps some of the key considerations discussed in the **Java** and **Enterprise** sections. Question numbers are used for cross-referencing with **Java** and **Enterprise** sections.
- **Q11** & **Q13** are discussed in more detail and can be used as a quick reference guide in a software project. All the other questions excluding **Q11** & **Q13** can be read just before an interview.

Q 01: How would you go about documenting your Java/J2EE application?
A 01: To be successful with a Java/J2EE project, proper documentation is vital.

- Before embarking on coding get the business requirements down. Build a complete list of requested features, sample screen shots (if available), use case diagrams, business rules etc as a **functional specification** document. This is the phase where business analysts and developers will be asking questions about user interface requirements, data tier integration requirements, use cases etc. Also prioritize the features based on the business goals, lead-times and iterations required for implementation.

- Prepare a **technical specification** document based on the functional specification. The technical specification document should cover:

  - **Purpose of the document**: This document will emphasise the customer service functionality …

  - **Overview**: This section basically covers background information, scope, any inclusions and/or exclusions, referenced documents etc.

  - **Basic architecture**: discusses or references baseline architecture document. Answers questions like Will it scale? Can this performance be improved? Is it extendable and/or maintainable? Are there any security issues? Describe the vertical slices to be used in the early iterations, and the concepts to be proved by each slice. Etc. **For example** which MVC [model-1, model-2 etc] paradigms (Refer **Q3** in Enterprise section for MVC) should we use? Should we use Struts, JSF, Spring etc or build our own framework? Should we use a business delegate (Refer **Q83** in Enterprise section for business delegate) to decouple middle tier with the client tier? Should we use AOP (Aspect Oriented Programming) (Refer **Q3** in Emerging Technologies/Frameworks)? Should we use dependency injection? Should we use annotations? Do we require internationalization? Etc.

  - **Assumptions, Dependencies, Risks** and **Issues**: highlight all the assumptions, dependencies, risks and issues. For example list all the risks you can identify.

  - **Design alternatives** for each key functional requirement. Also discuss why a particular design alternative was chosen over the others. This process will encourage developers analyse the possible design alternatives without having to jump at the obvious solution, which might not always be the best one.

  - **Processing logic:** discuss the processing logic for the client tier, middle tier and the data tier. Where required add process flow diagrams. Add any pre-process conditions and/or post-process conditions. (Refer **Q9** in Java section for design by contract).

  - **UML diagrams** to communicate the design to the fellow developers, solution designers, architects etc. Usually class diagrams and sequence diagrams are required. The other diagrams may be added for any special cases like (Refer **Q107** in Enterprise section):

    - **State chart diagram:** useful to describe behaviour of an object across several usecases.

    - **Activity diagram:** useful to express complex operations**.** Supports and encourages parallel behaviour. Activity and statechart diagrams are beneficial for workflow modelling with multi threaded programming.

    - **Collaboration and Sequence diagrams:** Use a collaboration or sequence diagram when you want to look at behaviour of several objects within a single use case. If you want to look at a single object across multiple use cases then use statechart.

    - **Object diagrams**: The Object diagrams show instances instead of classes. They are useful for explaining some complicated objects in detail such as highlighting recursive relationships etc.

  - **List the package names, class names, database names** and **table names** with a brief description of their responsibility in a tabular form.

- Prepare a **coding standards** document for the whole team to promote consistency and efficiency. Some coding practices can degrade performance for example:

  - Inappropriate use of String class. Use StringBuffer instead of String for compute intensive mutations (Refer **Q17** in Java section).

- Code in terms of interface. For example you might decide the LinkedList is the best choice for some application, but then later decide ArrayList might be a better choice. (Refer **Q15** in Java section)

  Wrong approach → ArrayList list = new ArrayList();
  Right approach → List list = new ArrayList(100)

- Set the initial capacity of a collection appropriately (e.g. ArrayList, HashMap etc). (Refer **Q15** in Java section).

- To promote consistency define standards for variable names, method names, use of logging, curly bracket positions etc.

- Prepare a **code review** document and templates for the whole team. Let us look at some of the elements the code review should cover:

  - **Proper variable declaration:** e.g. instance versus static variables, constants etc.

  - **Performance issues**: e.g. Use ArrayList, HashMap etc instead of Vector, Hashtable when there is no thread-safety issue.

  - **Memory issues:** e.g. Improper instantiation of objects instead of object reuse and object pooling, not closing valuable resource in a finally block etc.

  - **Thread-safety issues:** e.g. Java API classes like SimpleDateFormat, Calendar, DecimalFormat etc are not thread safe, declaring variables in JSP is not thread safe, storing state information in Struts action class or multi-threaded servlet is not thread safe.

  - **Error handling:** e.g. Re-throwing exception without nesting original exception, EJB methods not throwing EJB exception for system exceptions, etc.

  - **Use of coding standards:** e.g. not using frameworks, System.out is used instead of log4j etc.

  - **Design issues:** No re-use of code**,** no clear separation of responsibility, invalid use of inheritance to get method reuse, servlets performing JDBC direct access instead of using DAO (Data Access Objects) classes, HTML code in Struts action or servlet classes, servlets used as utility classes rather than as a flow controller etc.

  - **Documentation of code:** e.g. No comments, no header files etc

  - **Bugs:** e.g. Calling *setAutoCommit* within container-managed transaction, binary OR "|" used instead of logical OR "||", relying on pass-by-reference in EJB remote calls, *ResultSet* not being closed on exceptions, EJB methods not throwing *EJBException* for system exceptions etc (Refer **Q76** & **Q77** in Enterprise section)

- Prepare additional optional **guideline documents** as per requirements to be shared by the team. This will promote consistency and standards. For example:

  - Guidelines to setting up J2EE development environment.
  - Guidelines to version control system (CVS, VSS etc).
  - Guidelines to deployment steps, environment settings, ant targets etc.
  - Guidelines for the data modelling (any company standards).
  - Guidelines for error handling (Refer **Q34, Q35** in Java section & **Q76, Q77** in Enterprise section).
  - Guidelines for user interface design.
  - Project overview document.
  - Software development process document etc.

Some of the above mentioned documents, which are shared by the whole team, can be published in an internal website like Wiki. Wiki is a piece of server software that allows users to freely create and edit Web page content using any Web browser.

Q 02: How would you go about designing a Java/J2EE application?

A 02: Design should be specific to a problem but also should be general enough to address future requirements. Designing reusable object oriented software involves decomposing the business use cases into relevant objects and converting objects into classes.

- Create a **tiered architecture**: client tier, business tier and data tier. Each tier can be further logically divided into layers (Refer **Q2, Q3** on Enterprise section). Use MVC (**M**odel **V**iew **C**ontroller architecture for the J2EE and Java based GUI applications).

- Create a **data model**: A data model is a detailed specification of data oriented structures. This is different from the class modelling because it focuses solely on data whereas class models allow you to define both data and behaviour. **Conceptual data models** (aka domain models) are used to explore domain concepts with project stakeholders. **Logical data models** are used to explore the domain concepts, and their relationships. Logical data models depict entity types, data attributes end entity relationships (with Entity Relationship (ER) diagrams). **Physical data models** are used to design the internal schema of a database depicting the tables, columns, and the relationships between the tables. Data models can be created by performing the following tasks:

  - **Identify entity types, attributes and relationships**: use entity relationship (E-R) diagrams.

  - **Apply naming conventions (e.g. for tables, attributes, indices, constraints etc)**: Your organization should have standards and guidelines applicable to data modelling.

  - **Assign keys**: surrogate keys (e.g. assigned by the database like Oracle sequences etc, max()+1, universally unique identifiers UUIDs, etc), natural keys (e.g. Tax File Numbers, Social Security Numbers etc), and composite keys.

  - **Normalize to reduce data redundancy and denormalize to improve performance**: Normalized data have the advantage of information being stored in one place only, reducing the possibility of inconsistent data. Furthermore, highly normalized data are loosely coupled. But normalization comes at a performance cost because to determine a piece of information you have to join multiple tables whereas in a denormalized approach the same piece of information can be retrieved from a single row of a table. Denormalization should be used only when performance testing shows that you need to improve database access time for some of your tables.

Note: Creating a data model (logical, physical etc) before design model is a matter of preference, but many OO methodologies are based on creating the data model from the object **design model** (i.e. you may need to do some work to create an explicit data model but only after you have a complete OO domain and design model). In many cases when using ORM tools like Hibernate, you do not create the data model at all.

- Create a **design model**: A design model is a detailed specification of the objects and relationships between the objects as well as their behaviour. (Refer **Q107** on Enterprise section)

  - **Class diagram:** contains the implementation view of the entities in the design model. The design model also contains core business classes and non-core business classes like persistent storage, security management, utility classes etc. The class diagrams also describe the structural relationships between the objects.

  - **Use case realizations:** are described in sequence and collaboration diagrams.

- **Design considerations when decomposing business use cases into relevant classes**: designing reusable and flexible design models requires the following considerations:

  - **Granularity of the objects** (fine-grained, coarse-grained etc): Can we minimise the network trip by passing a coarse-grained value object instead of making 4 network trips with fine-grained parameters? (Refer **Q85** in Enterprise section). Should we use method level (coarse-grained) or code level (fine-grained) thread synchronization? (Refer **Q40** in Java section). Should we use a page level access security or a fine-grained programmatic security?

  - **Coupling between objects** (loosely coupled versus strongly coupled). Should we use business delegate pattern to loosely couple client and business tier? (Refer **Q83** in Enterprise section) or Should we use dependency injection? (Refer **Q09** in Emerging Technologies/Frameworks).

  - **Network overheads** for remote objects like EJB, RMI etc: Should we use the session façade, value object patterns? (Refer **Q84** & **Q85** in Enterprise section).

- **Definition of class interfaces** and **inheritance hierarchy:** Should we use an abstract class or an interface? Is there any common functionality that we can move to the super class (or parent class)? Should we use interface inheritance with object composition for code reuse as opposed to implementation inheritance? Etc. (Refer **Q8**, **Q10** in Java section).

- **Establishing key relationships** (aggregation, composition, association etc): Should we use aggregation or composition? [composition may require cascade delete] (Refer **Q107, Q108** in Enterprise section – under class diagrams). Should we use an "**is a**" (generalization) relationship or a "**has a**" (composition) relationship? (Refer **Q7** in Java section).

- **Applying polymorphism** and **encapsulation**: Should we hide the member variables to improve integrity and security? (Refer **Q8** in Java section). Can we get a polymorphic behaviour so that we can easily add new classes in the future? (Refer **Q8** in Java section).

- **Applying well-proven design patterns** (like Gang of four design patterns, J2EE design patterns, EJB design patterns etc) help designers to base new designs on prior experience. Design patterns also help you to choose design alternatives (Refer **Q11** in How would you go about…).

- **Scalability** of the system: **Vertical scaling** is achieved by increasing the number of servers running on a single machine. **Horizontal scaling** is achieved by increasing the number of machines in the cluster. Horizontal scaling is more reliable than the vertical scaling because there are multiple machines involved in the cluster. In vertical scaling the number of server instances that can be run on one machine are determined by the CPU usage and the JVM heap memory.

- **How do we replicate the session state**? Should we use stateful session beans or HTTP session? Should we serialize this object so that it can be replicated?

- **Internationalization** requirements for multi-language support: Should we support other languages? Should we support multi-byte characters in the database?

- **Vertical slicing**: Getting the reusable and flexible design the first time is impossible. By developing the initial **vertical slice** of your design you eliminate any nasty integration issues later in your project. Also get the design patterns right early on by building the vertical slice. It will give you experience with what does work and what does not work with Java/J2EE. Once you are happy with the initial vertical slice then you can apply it across the application. The initial vertical slice should be based on a typical business use case. Refer **Q136** in Enterprise section.

- **Ensure the system is configurable** through property files, xml descriptor files, annotations etc. This will improve flexibility and maintainability. Avoid hard coding any values. Use a constant class for values, which rarely change and use property files, xml descriptor files, annotations etc for values, which can change more frequently (e.g. process flow steps etc) and/or environment related configurations(e.g. server name, server port, LDAP server location etc).

- **Design considerations during design, development and deployment phases:** designing a fast, secured, reliable, robust, reusable and flexible system require considerations in the following key areas:

  - **Performance issues** (network overheads, quality of the code etc): Can I make a single coarse-grained network call to my remote object instead of 3 fine-grained calls?

  - **Concurrency issues (multi-threading etc)**: What if two threads access my object simultaneously will it corrupt the state of my object?

  - **Transactional issues (ACID properties)**: What if two clients access the same data simultaneously? What if one part of the transaction fails, do we rollback the whole transaction? What if the client resubmits the same transactional page again?

  - **Security issues**: Are there any potential security holes for SQL injection or URL injection by hackers?

  - **Memory issues**: Is there any potential memory leak problems? Have we allocated enough heap size for the JVM? Have we got enough perm space allocated since we are using 3rd party libraries, which generate classes dynamically? (e.g. JAXB, XSLT, JasperReports etc)

  - **Scalability issues**: Will this application scale vertically and horizontally if the load increases? Should this object be serializable? Does this object get stored in the HttpSession?

- **Maintainability, reuse, extensibility etc**: How can we make the software reusable, maintainable and extensible? What design patterns can we use? How often do we have to refactor our code?
- **Logging and auditing** if something goes wrong can we look at the logs to determine the root cause of the problem?
- **Object life cycles**: Can the objects within the server be created, destroyed, activated or passivated depending on the memory usage on the server? (e.g. EJB).
- **Resource pooling**: Creating and destroying valuable resources like database connections, threads etc can be expensive. So if a client is not using a resource can it be returned to a pool to be reused when other clients connect? What is the optimum pool size?
- **Caching** can we save network trips by storing the data in the server's memory? How often do we have to clear the cache to prevent the in memory data from becoming stale?
- **Load balancing**: Can we redirect the users to a server with the lightest load if the other server is overloaded?
- **Transparent fail over**: If one server crashes can the clients be routed to another server without any interruptions?
- **Clustering**: What if the server maintains a state when it crashes? Is this state replicated across the other servers?
- **Back-end integration**: How do we connect to the databases and/or legacy systems?
- **Clean shutdown**: Can we shut down the server without affecting the clients who are currently using the system?
- **Systems management**: In the event of a catastrophic system failure who is monitoring the system? Any alerts or alarms? Should we use JMX? Should we use any performance monitoring tools like Tivoli etc?
- **Dynamic redeployment**: How do we perform the software deployment while the site is running? (Mainly for mission critical applications 24hrs X 7days).
- **Portability issues**: Can I port this application to a different server 2 years from now?

Q 03: How would you go about identifying performance and/or memory issues in your Java/J2EE application?

A 03: Profiling can be used to identify any performance issues or memory leaks. Profiling can identify what lines of code the program is spending the most time in? What call or invocation paths are used to reach at these lines? What kinds of objects are sitting in the heap? Where is the memory leak? Etc.

- There are many tools available for the optimization of Java code like **JProfiler, Borland OptimizeIt** etc. These tools are very powerful and easy to use. They also produce various reports with graphs.

  Optimizeit™ Request Analyzer provides advanced profiling techniques that allow developers to analyse the performance behaviour of code across J2EE application tiers. Developers can efficiently prioritize the performance of Web requests, JDBC, JMS, JNDI, JSP, RMI, and EJB so that trouble spots can be proactively isolated earlier in the development lifecycle.

  Thread Debugger tools can be used to identify threading issues like thread starvation and contention issues that can lead to system crash.

  Code coverage tools can assist developers with identifying and removing any dead code from the applications.

- **Hprof** which comes with JDK for free. Simple tool.

```
Java –Xprof myClass

java  -Xrunhprof:[help]|[<option>=<value>]
java  -Xrunhprof:cpu=samples, depth=6, heap=sites
```

- Use operating system process monitors like NT/XP Task Manager on PCs and commands like ps, iostat, netstat, vmstat, uptime, nfsstat etc on UNIX machines.

- Write your own wrapper MemoryLogger and/or PerformanceLogger utility classes with the help of **totalMemory()** and **freeMemory()** methods in the Java **Runtime** class for memory usage and **System.currentTimeMillis()** method for performance. You can place these MemoryLogger and PerformanceLogger calls strategically in your code. Even better approach than utility classes is using Aspect Oriented Programming (AOP) for pre and post memory and/or performance recording where you have the control of activating memory/performance measurement only when needed. (Refer **Q3 – Q5** in Emerging Technologies/Frameworks section).

Q 04: How would you go about minimising memory leaks in your Java/J2EE application?

A 04: Java's' memory management (i.e. Garbage Collection) prevents lost references and dangling references but it is still possible to create memory leaks in other ways. If the application runs with memory leaks for a long duration you will get the error **java.lang.OutOfMemoryError**.

In Java, typically the memory leak occurs when **an object of a longer lifecycle has a reference to the objects of a short life cycle**. This prevents the objects with short life cycle being garbage collected. The developer must remember to remove the reference to the short-lived objects from the long-lived objects. Objects with the same life cycle do not cause any problem because the garbage collector is smart enough to deal with the circular references (Refer **Q33** in Java section).

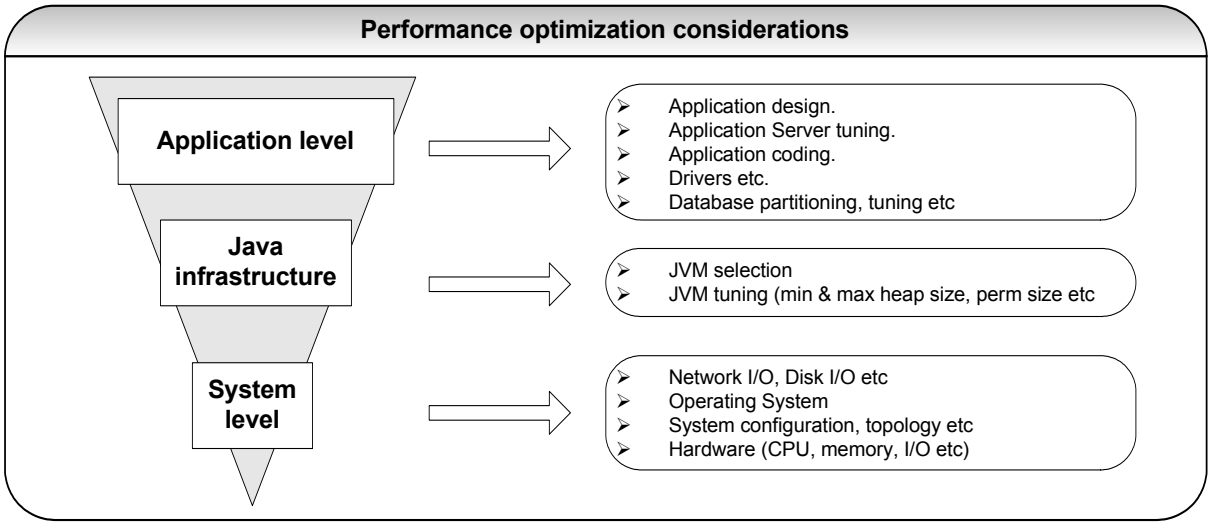
- Java collection classes like Hashtable, ArrayList etc maintain references to other objects. So having a long life cycle ArrayList pointing to many short-life cycle objects can cause memory leaks.

- Commonly used **singleton design pattern** (Refer **Q45** in Java section) can cause memory leaks. Singletons typically have a long life cycle. If a singleton has an ArrayList or a Hashtable then there is a potential for memory leaks.

- Java programming language includes a finalize method that allows an object to free system resources, in other words, to clean up after itself. However using finalize doesn't guarantee that a class will clean up resources expediently. A better approach for cleaning up resources involves the finally method and an explicit close statement. So freeing up the valuable resource in the finalize method or try {} block instead of finally {} block can cause memory leaks (Refer **Q45** in Enterprise section).

Q 05: How would you go about improving performance in your Java/J2EE application?

A 05: The performance bottlenecks can be attributed to one or more of the following:



Let us discuss some of the aspects in detail:

- Java/J2EE application code related performance bottlenecks:

  - Refer **Q63** in Java section.

- ❖ Refer **Q123, Q125** in Enterprise section.

- Java/J2EE design related performance bottlenecks. Application design is one of the most important considerations for performance. A well-designed application will not only avoid many performance pitfalls but will also be easier to maintain and modify during the performance-testing phase of the project.

  - ❖ Use proper design patterns to **minimise network trips** (session facade, value object Refer etc **Q83-Q87** in Enterprise section).

  - ❖ **Minimise serialization cost** by implementing session beans with remote interfaces and entity beans with local interfaces (applicable to EJB 2.x) or even the session beans can be implemented with local interfaces sharing the same JVM with the Web tier components. For EJB1.x some EJB containers can be configured to use **pass-by-reference** instead of pass-by-value (pass-by-value requires serialization) Refer **Q69, Q82** in Enterprise section.

  - ❖ Use of multi-threading from a thread-pool (say 10 – 50 threads). Using a large number of threads adversely affects performance by consuming memory through thread stacks and CPU by context switching.

- Database related performance bottlenecks.

  - ❖ Use proper database indexes. Numeric indices are more efficient than character based indices. Minimise the number of columns in your composite keys. Performing a number of "INSERT" operations is more efficient when fewer columns are indexed and "SELECT" operations are more efficient when, adequately indexed based on columns frequently used in your "WHERE" clause. So it is a trade-off between "SELECT" and "INSERT" operations.

  - ❖ Minimise use of composite keys or use fewer columns in your composite keys.

  - ❖ Partition the database for performance based on the most frequently accessed data and least frequently accessed data.

  - ❖ Identify and optimise your SQL queries causing performance problems (Refer **Q97** in Enterprise section).

  - ❖ De-normalise your tables where necessary for performance (Refer **Q98** in Enterprise section).

  - ❖ Close database connections in your Java code in the finally block to avoid any 'open cursors' problem (Refer **Q45** in Enterprise section).

  - ❖ Use **optimistic concurrency** as opposed to **pessimistic concurrency** where appropriate (Refer **Q78** in Enterprise section).

- Application Server, JVM, Operating System, and/or hardware related performance bottlenecks.

  - ❖ **Application Server**: Configure the application server for optimum performance (Refer **Q88**, **Q123** in Enterprise section).

  - ❖ **Operating System**: Check for any other processes clogging up the system resources, maximum number of processes it can support or connect, optimise operating system etc.

  - ❖ **Hardware**: Insufficient memory, insufficient CPU, insufficient I/O, limitation of hardware configurations, network constraints like bandwidth, message rates etc**.**

Q 06: How would you go about identifying any potential thread-safety issues in your Java/J2EE application?

A 06: When you are writing graphical programs like Swing or Internet programs using servlets or JSPs multi-threading is a necessity for all but some special and/or trivial programs.

An application program or a process can have multiple threads like multiple processes that can run on one computer. The multiple threads appear to be doing their work in parallel. When implemented on a multi-processor machine, they can actually work in parallel.

Unlike processes, threads share the same address space (Refer **Q36** in Java section) which means they can read and write the same variables and data structures. So care should be taken to avoid one thread disturbing the work of another thread. Let us look at some of the common situations where care should be taken:

- Swing components can only be accessed by one thread at a time. A few operations are guaranteed to be thread safe but the most others are not. Generally the Swing components should be accessed through an ***event-dispatching thread***. (Refer **Q53** in Java section).

- A typical Servlet life cycle creates a single instance of each servlet and creates multiple threads to handle the service() method. **The multi-threading aids efficiency but the servlet code must be coded in a thread safe manner.** The shared resources (e.g. instance variable) should be appropriately synchronized or should only use variables in a read-only manner. (Refer **Q16** in Enterprise section).

- The declaration of variables in JSP is not thread-safe, because the declared variables end up in the generated servlet as an instance variable, not within the body of the _jspservice() method. (Refer **Q34** in Enterprise section).

- Struts framework action classes are not thread-safe. (Refer **Q113** in Enterprise section).

- Some Java collection classes like Hashmap, ArrayList etc are not thread-safe. (Refer **Q13** in Java section).

- Some of the Java core library classes are not thread safe. For e.g. java.util.SimpleDateFormat, java.util.Locale etc.

Q 07: How would you go about identifying any potential transactional issues in your Java/J2EE application?
A 07:

- When a connection is created, it is in **auto-commit** mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed immediately after it is executed. The way to allow two or more statements to be grouped into a transaction is to **disable** auto-commit mode. (Refer **Q43** in Enterprise section). Disabling auto-commit mode can improve performance by minimising number of times it accesses the database.

- A transaction is often described by ACID properties (Atomic, Consistent, Isolated and Durable). A **distributed transaction** is an ACID transaction between two or more independent transactional resources like two separate databases. For a transaction to commit successfully, all of the individual resources must commit successfully. If any of them are unsuccessful, the transaction must rollback in all of the resources. A **2-phase commit** is an approach for committing a distributed transaction in 2 phases. Refer **Q73** in Enterprise section.

- Isolation levels provide a degree of control of the effects one transaction can have on another concurrent transaction. Concurrent effects are determined by the precise ways in which, a particular relational database handles locks and its drivers may handle these locks differently. Isolation levels are used to overcome transactional problems like lost update, uncommitted data (aka dirty reads), inconsistent data (aka. phantom update), and phantom insert. Higher isolation levels can adversely affect performance at the expense of data accuracy. Refer **Q72** in Enterprise section.

Isolation Level	Lost Update	Uncommitted Data	Inconsistent Data	Phantom Insert
Read Uncommitted	Prevented by DBMS	Can happen	Can happen	Can happen
Read Committed	Prevented by DBMS	Prevented by DBMS	Can happen	Can happen
Repeatable Read	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS	Can happen
Serializable	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS	Prevented by DBMS

- Decide between optimistic and pessimistic concurrency control. (Refer **Q78** in Enterprise section).

- Evaluate a strategy to determine if the data is stale when using strategies to cache data. (Refer **Q79** in Enterprise section).

EJB related transactional issues:

- Set the appropriate transactional attributes for the EJBs. (Refer **Q71** in Enterprise section).

- Set the appropriate isolation level for the EJB. The isolation level should not be any more restrictive than it has to be. Higher isolation levels can adversely affect performance. (Refer **Q72** in Enterprise section). Isolation levels are application server specific and not part of the standard EJB configuration.

- In EJB 2.x, transactions are rolled back by the container when a **system exception** is thrown. When an **application exception** is thrown then the transactions are not rolled back by the container. So the developer has to roll it back using **ctx.setRollbackOnly()** call. (Refer **Q76, Q77** in Enterprise section).

- Detect doomed transactions to avoid performing any unnecessary compute intensive operations. (Refer **Q72** in Enterprise section).

Q 08: How would you go about applying the Object Oriented (OO) design concepts in your Java/J2EE application?
A 08:

<table>
<tr><th>Question</th><th>Answer</th></tr>
<tr><td>What are the key characteristics of an OO language like Java?</td><td>A true object oriented language should support the following 3 characteristics:

❖ Encapsulation (aka information hiding): implements information hiding and modularity (abstraction).

❖ Polymorphism: The same message sent to different objects, results in behaviour that is dependent on the nature of the object receiving the message.

❖ Inheritance: Encourages code reuse and code organisation by defining the new class based on the existing class.

What is dynamic binding?

Dynamic binding (aka late binding): The dynamic binding is used to implement polymorphism. Objects could come from local process or from across the network from a remote process. We should be able to send messages to objects without having to know their types at the time of writing the code. Dynamic binding provides maximum flexibility at the execution time. Usually dynamic binding or late binding takes a small performance hit.

Refer Q8 in Java section.

Let us take an example to illustrate dynamic binding through polymorphic behaviour:

Say you have a method in Java

<pre>void draw(Shape s) {
  s.erase();
  // ...
  s.draw();
}</pre>
The above method will talk to any shape, so it is independent of the specific type of object it is erasing and drawing. Now let us look at some other program, which is making use of this draw(Shape s) method:

<pre>Circle cir = new Circle();
Square sq = new Square();

draw(cir);
draw(sq);</pre>
So the interesting thing is that the method call to draw(Shape s) will cause different code to be executed. So you send a message to an object even though you don't know what specific type it is and the right thing happens. This is called dynamic binding, which gives you polymorphic behaviour.</td></tr>
<tr><td>How will you decide whether to use an interface or an abstract class?</td><td>❖ Abstract Class: Often in a design, you want the base class to present only an interface for its derived classes. That is, you don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by making that class abstract using the abstract key word. If anyone tries to make an object of an abstract class, the compiler prevents them. This is a tool to enforce a particular design.

❖ Interface: The interface key word takes the concept of an abstract class one step further by preventing any function definitions at all. An interface is a very useful and commonly used tool, as it provides the perfect separation of interface and implementation. In addition, you can combine many interfaces together, if you wish. (You cannot inherit from more than one regular class or abstract class.)

Now the design decision…</td></tr>
</table>

<table>
<tr><td></td><td>
When to use an Abstract Class: Abstract classes are excellent candidates inside of application frameworks. Abstract classes let you define some default behaviour and force subclasses to provide any specific behaviour.

When to use an Interface: If you need to change your design frequently, I prefer using interface to abstract. For example, the strategy pattern lets you swap new algorithms and processes into your program without altering the objects that use them. Example: Strategy Design Pattern.

Another justification of interfaces is that they solved the 'diamond problem' of traditional multiple inheritance. Java does not support multiple inheritances. Java only supports multiple interface inheritance. Interface will solve all the ambiguities caused by this 'diamond problem'. Refer Q10 in Java section.

Interface inheritance vs. Implementation inheritance: Prefer interface inheritance to implementation inheritance because it promotes the design concept of coding to an interface and reduces coupling. Interface inheritance can achieve code reuse with the help of object composition. Refer Q08 in Java section.
</td></tr>
<tr><td>Why abstraction is important in Object Oriented programming?</td><td>
The software you develop should optimally cater for the current requirements and problems and also should be flexible enough to easily handle future changes.

Abstraction is an important OO concept. The ability for a program to ignore some aspects of the information that it is manipulating, ie. Ability to focus on the essential. Each object in the system serves as a model of an abstract "actor" that can perform work, report on and change its state, and "communicate" with other objects in the system, without revealing how these features are implemented. Abstraction is the process where ideas are distanced from the concrete implementation of the objects. The concrete implementation will change but the abstract layer will remain the same.

Let us look at an analogy:

When you drive your car you do not have to be concerned with the exact internal working of your car (unless you are a mechanic). What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Over the years a car's engine has improved a lot but its basic interface has not changed (ie you still use steering wheel, brake pedal, accelerator pedal etc to interact with your car). This means that the implementation has changed over the years but the interface remains the same. Hence the knowledge you have of your car is abstract.
</td></tr>
<tr><td>Explain black-box reuse and white-box reuse? Should you favour Inheritance (white-box reuse) or aggregation (black-box reuse)?</td><td>
Black-box reuse is when a class uses another class without knowing the internal contents of it. The black-box reuses are:

❖ Dependency is the weakest type of black-box reuse.
❖ Association is when one object knows about or has a relationship with the other objects.
❖ Aggregation is the whole part relationship where one object contains one or more of the other objects.
❖ Composition is a stronger whole part relationship

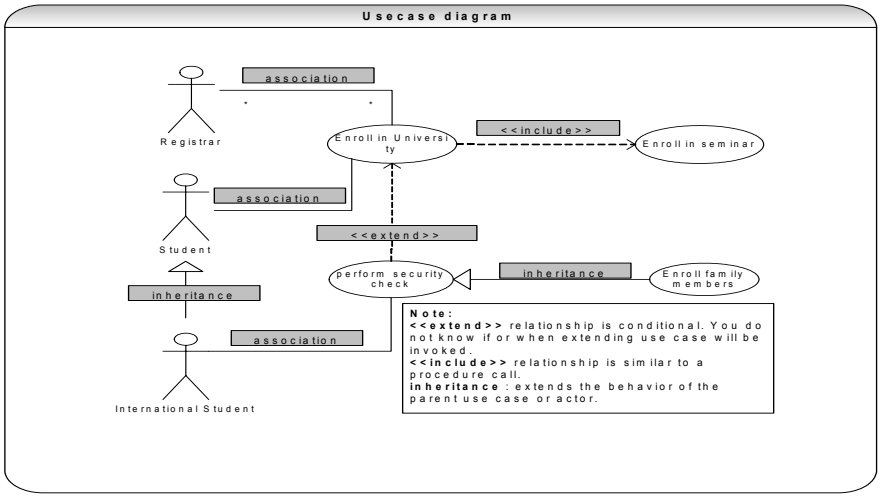
Refer Q107, Q108 in Enterprise section

White-box reuse is when a class knows internal contents of another class. E.g. inheritance is used to modify implementation for reusability.
<table>
<tr><th>Aggregation (Black-box reuse)</th><th>Inheritance (White-box reuse)</th></tr>
<tr><td>Defined dynamically or at run time via object references. Since only interfaces are used, it has the advantage of maintaining the integrity (ie encapsulation).</td><td>Inheritance is defined statically or at compile time. Inheritance allows an easy way to modify implementation for reusability.</td></tr>
<tr><td>Disadvantage of aggregation is that it increases the number of objects and relationships.</td><td>A disadvantage of inheritance is that it breaks encapsulation, which implies implementation dependency. This means when you want to carry out the redesign the super class (ie parent class) has to be modified or replaced which is more likely to affect the subclasses as well. In general it will affect the whole inheritance hierarchy.

Verdict: So the tendency is to favour aggregation over inheritance.</td></tr>
</table>
</td></tr>
</table>

What is your understanding on Aspect Oriented Programming (AOP)?	Aspect-Oriented Programming (**AOP**) complements **OO** programming by allowing developers to dynamically modify the static OO model to create a system that can grow to meet new requirements. AOP allows us to dynamically modify our static model to include the code required to fulfil the secondary requirements (like auditing, logging, security, exception handling etc) without having to modify the original static model (in fact, we don't even need to have the original code). Better still, we can often keep this additional code in a single location rather than having to scatter it across the existing model, as we would have to if we were using OO on its own. (Refer **Q3 –Q5** in Emerging Technologies/Frameworks section.) **For example** A typical Web application will require a servlet to bind the HTTP request to an object and then passes to the business handler object to be processed and finally return the response back to the user. So initially only a minimum amount of code is required. But once you start adding all the other additional secondary requirements (aka **crosscutting concerns**) like logging, auditing, security, exception-handling etc the code will inflate to 2-4 times its original size. This is where AOP can help.

Q 09: How would you go about applying the UML diagrams in your Java/J2EE project?
A 09:

Question	Answer
Explain the key relationships in the use case diagrams?	Refer **Q107** in Enterprise section. Use case has 4 types of relationships: **Between actor and use case** ❖ **Association**: Between **actor and use case**. May be navigable in both directions according to the initiator of the communication between the actor and the usecase. **Between use cases** ❖ **Extends**: This is an optional extended behaviour of a use case. This behaviour is executed only under certain conditions such as performing a security check etc. ❖ **Includes:** This specifies that the base use case needs an additional use case to fully describe its process. It is mainly used to show common functionality that is shared by several use cases. ❖ **Inheritance (or generalization)**: Child use case inherits the behaviour of its parent. The child may override or add to the behaviour of the parent. 
What is the main difference between the collaboration diagram and the sequence diagram?	Refer **Q107** in Enterprise section: Collaboration diagrams convey the same message as sequence diagrams but the collaboration diagrams focus on **object roles** instead of **times in which** the messages are sent. The sequence diagram is time line driven.
When to use various UML diagrams?	Refer **Q107** in Enterprise section. ❖ **Use case diagrams**: ▪ Determining the user requirements. New use cases often generate new requirements. ▪ Communicating with clients. The simplicity of the diagram makes use case diagrams a good way for designers and developers to communicate with clients. ▪ Generating test cases. Each scenario for the use case may suggest a suite of test

	cases. ❖ **Class diagrams**: ▪ Class diagrams are the backbone of **O**bject **O**riented methods. So they are used frequently. ▪ Class diagrams can have a conceptual perspective and an implementation perspective. During the analysis draw the conceptual model and during implementation draw the implementation model. ❖ **Interaction diagrams (Sequence and/or Collaboration diagrams):** ▪ When you want to look at behaviour of **several objects within a single use case**. If you want to look at a **single object across multiple use cases** then use statechart diagram as described below. ❖ **State chart diagrams:** ▪ Statechart diagrams are good at describing the **behaviour of an object across several use cases**. But they are not good at describing the interaction or collaboration between many objects. Use interaction and/or activity diagrams in conjunction with the statechart diagram to communicate complex operations involving multi-threaded programs etc. ▪ Use it only for classes that have complex state changes and behaviour. **For example:** the User Interface (UI) control objects, Objects shared by multi-threaded programs etc. ❖ **Activity diagram**: ▪ **Activity** and **Statechart** diagrams are generally useful to express complex operations. The great strength of activity diagrams is that they support and encourage parallel behaviour. An activity and statechart diagrams are beneficial for workflow modelling with multi threaded programming.

Q 10: How would you go about describing the software development processes you are familiar with?

A 10: In addition to technical questions one should also have a good understanding of the software development process.

Question	Answer

What is the key difference between the waterfall approach and the iterative approach to software development? How to decide which one to use?	Refer **Q103 – Q105** in Enterprise section
	
	Waterfall approach is sequential in nature. The **iterative** approach is non-sequential and incremental. The iterative and incremental approach has been developed based on the following:
	
	• **You can't express all your needs up front.** It is usually not feasible to define in detail (that is, before starting full-scale development) the operational capabilities and functional characteristics of the entire system. These usually evolve over time as development progresses.
	
	• **Technology changes over time.** Some development lifecycle spans a long period of time during which, given the pace at which technology evolves, significant technological shifts may occur.
	
	• **Complex systems.** This means it is difficult to cope with them adequately unless you have an approach for mastering complexity.
	
	How to decide which one to use?
	
	Waterfall approach is more suitable in the following circumstances:
	
	• Have a small number of unknowns and risks. That is if
	  ◦ It has a known domain.
	  ◦ The team is experienced in current process and technology.
	  ◦ There is no new technology.
	  ◦ There is a pre-existing architecture baseline.
	• Is of short duration (two to three months).
	• Is an evolution of an existing system?
	
	The **iterative** approach is more suitable (Refer **Q136** in Enterprise Section)
	
	• Have a large number of unknowns and risks. So it pays to design, develop and test a vertical slice iteratively and then replicate it through other iterations. That is if

	• Integrating with new systems. • New technology and/or architecture. • The team is fairly keen to adapt to this new process. • Is of large duration (longer than 3 months). • Is a new system?
Have you used extreme programming techniques? Explain?	Extreme Programming (or XP) is a set of values, principles and practices for rapidly developing high-quality software that provides the highest value for the customer in the fastest way possible. **XP is a minimal instance of RUP**. XP is extreme in the sense that it takes **12 well-known software development "best practices" to their logical extremes.** The 12 core practices of XP are: 1. **The Planning Game**: Business and development cooperate to produce the maximum business value as rapidly as possible. The planning game happens at various scales, but the basic rules are always the same: ▪ Business comes up with a list of desired features for the system. Each feature is written out as a **user story** (or PowerPoint screen shots with changes highlighted), which gives the feature a name, and describes in broad strokes what is required. User stories are typically written on 4x6 cards. ▪ Development team estimates how much effort each story will take, and how much effort the team can produce in a given time interval (i.e. the iteration). ▪ Business then decides which stories to implement in what order, as well as when and how often to produce production releases of the system. 2. **Small releases**: Start with the smallest useful feature set. Release early and often, adding a few features each time. 3. **System metaphor**: Each project has an organising metaphor, which provides an easy to remember naming convention. 4. **Simple design**: Always use the simplest possible design that gets the job done. The requirements will change tomorrow, so only do what's needed to meet today's requirements. 5. **Continuous testing**: Before programmers add a feature, they write a test for it. Tests in XP come in two basic flavours. ▪ **Unit tests** are automated tests written by the developers to test functionality as they write it. Each unit test typically tests only a single class, or a small cluster of classes. Unit tests are typically written using a unit-testing framework, such as **JUnit**. ▪ **Customer to test that the overall system is functioning as specified, defines acceptance tests (aka Functional tests)**. Acceptance tests typically test the entire system, or some large chunk of it. When all the acceptance tests pass for a given user story, that story is considered complete. At the very least, an acceptance test could consist of a script of user interface actions and expected results that a human can run. Ideally acceptance tests should be automated using frameworks like Canoo Web test, Selenium Web test etc. 6. **Refactoring**: Refactor out any duplicate code generated in a coding session. You can do this with confidence that you didn't break anything because you have the tests. 7. **Pair Programming**: All production code is written by two programmers sitting at one machine. Essentially, all code is reviewed as it is written. 8. **Collective code ownership**: No single person "owns" a module. Any developer is expected to be able to work on any part of codebase at any time. 9. **Continuous integration**: All changes are integrated into codebase at least daily. The tests have to run 100% both before and after integration. You can use tools like Ant, CruiseControl, and Maven etc to continuously build and integrate your code. 10. **40-Hour Workweek**: Programmers go home on time. In crunch mode, up to one week of overtime is allowed. But multiple consecutive weeks of overtime are treated as a sign that something is very wrong with the process. 11. **On-site customer**: Development team has continuous access to a real live customer or business owner, that is, someone who will actually be using the system. For commercial software with lots of customers, a customer proxy (usually the product manager, Business

	Analyst etc) is used instead. 12. **Coding standards**: Everyone codes to the same standards. Ideally, you shouldn't be able to tell by looking at it, which developer on the team has touched a specific piece of code. A typical extreme programming project will have: • All the programmers in a room together usually sitting around a large table. • Fixed number of iterations where **each iteration takes 1-3 weeks**. At the beginning of each iteration get together with the customer. • Pair-programming. • Writing test cases first (i.e. TDD). • Delivery of a functional system at the end of 1-3 week iteration.
Have you used agile (i.e. Lightweight) software development methodologies?	**Agile (i.e. lightweight) software development process** is gaining popularity and momentum across organizations. Several methodologies fit under this agile development methodology banner. All these methodologies share many characteristics like **iterative and incremental development**, **test driven development** (i.e. TDD), **stand up meetings to improve communication**, **automatic testing, build and continuous integration of code** etc. Among all the agile methodologies XP is the one which has got the most attention. Different companies use different flavours of agile methodologies by using different combinations of methodologies (e.g. primarily XP with other methodologies like Scrum, FDD, TDD etc). Refer **Q136** in Enterprise section.

Q 11: How would you go about applying the design patterns in your Java/J2EE application?
A 11: It is really worth reading books and articles on design patterns. It is sometimes hard to remember the design patterns, which you do not use regularly. So if you do not know a particular design pattern you can always honestly say that you have not used it and subsequently suggest that you can explain another design pattern, which you have used recently or more often. It is always challenging to decide, which design pattern to use when? How do you improve your design pattern skills? Practice, practice, practice. I have listed some of the design patterns below with scenarios and examples:

Note: To keep it simple, System.out.println(…) is used. In real practice, use logging frameworks like log4j. Also package constructs are not shown. In real practice, each class should be stored in their relevant packages like com.items etc. Feel free to try these code samples by typing them into a Java editor of your choice and run the main class *Shopping*. Also constants should be declared in a typesafe manner as shown below:

```
/**
 * use typesafe enum pattern as shown below if you are using below JDK 1.5 or use "enum" if you are using JDK 1.5
 */
public class ItemType {
    private final String name;

    public static final ItemType Book = new ItemType("book");
    public static final ItemType CD = new ItemType("cd");
    public static final ItemType COSMETICS = new ItemType("cosmetics");
    public static final ItemType CD_IMPORTED = new ItemType("cd_imported");

    private ItemType(String name) {this.name = name;}
    public String toString() {return name;}
    //add compareTo(), readResolve() methods etc as required ...
}
```

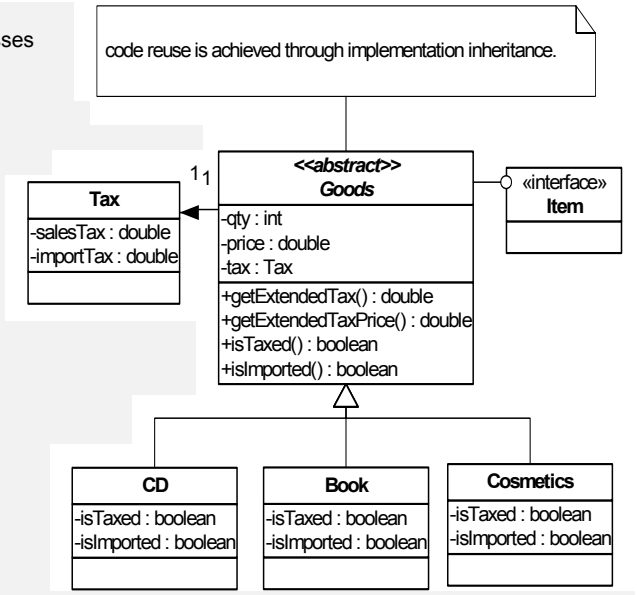
Scenario: A company named XYZ Retail is in the business of selling *Books*, *CDs* and *Cosmetics*. *Books* are sales tax exempt and *CDs* and *Cosmetics* have a sales tax of 10%. CDs can be imported and attracts an import tax of 5%. Write a shopping basket program, which will calculate extended price (qty * (unitprice + tax)) inclusive of tax for each item in the basket, total taxes and grand total.

Solution: Sample code for the items (i.e. Goods) sold by XYZ Retail. Let's define an *Item* interface to follow the design principle of **code to an interface not to an implementation**. **CO**

```
public interface Item {
    public static final int TYPE_BOOK = 1;
    public static final int TYPE_CD = 2;
    public static final int TYPE_COSMETICS = 3;
    public static final int TYPE_CD_IMPORTED = 4;

    public double getExtendedTax();
```

```
    public double getExtendedTaxPrice() throws ItemException;
    public void setImported(boolean b);
    public String getDescription();
}
```

The following class *Goods* cannot be instantiated (since it is **abstract**). You use this abstract class to achieve **code reuse**.




```
/**
 * abstract parent class, which promotes code reuse for all the subclasses
 * like Book, CD, and Cosmetics. implements interface Item to
 * promote design principle code to interface not to an implementation.
 */
public abstract class Goods implements Item {
  //define attributes
  private String description;
  private int qty;
  private double price;
  private Tax tax = new Tax();

  public Goods(String description, int qty, double price) {
     this.description = description;
     this.qty = qty;
     this.price = price;
  }

  protected abstract boolean isTaxed();
  protected abstract boolean isImported();

  public double getExtendedTax() {
       tax.calculate(isTaxed(), isImported(), price);
       return this.tax.getTotalUnitTax() * qty;
  }

  public double getExtendedTaxPrice() throws ItemException {
     if (tax == null) {
        throw new ItemException("Tax should be calculated first:");
     }
     return qty * (this.tax.getTotalUnitTax() + price);
  }

  //getters and setters go here for attributes like description etc
  public String getDescription() {
     return description;
  }

  public String toString() {
    return qty + " " + description + " : ";
  }
}
```

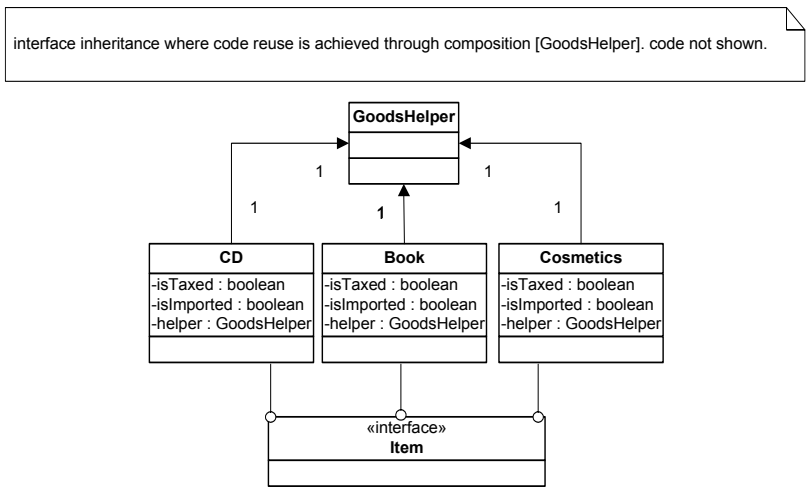
The *Book*, *CD* and *Cosmetics* classes can be written as shown below:

```
public class Book extends Goods {
    private boolean isTaxed = false;
    private boolean isImported = false;

  public Book(String description, int qty, double price) {
       super(description, qty, price);
  }

  public boolean isTaxed() {
       return isTaxed;
  }

  public boolean isImported() {
       return isImported;
  }

  public void setImported(boolean b) {
       isImported = b;
  }
}
```

```
public class CD extends Goods {
    private boolean isTaxed = true;
    private boolean isImported = false;

    public CD(String description, int qty, double price) {
       super(description, qty, price);
   }

   public boolean isTaxed() {
      return isTaxed;
   }

   public boolean isImported() {
      return isImported;
   }

   public void setImported(boolean b) {
      isImported = b;
   }
}
```

```
public class Cosmetics extends Goods {
    private boolean isTaxed = true;
    private boolean isImported = false;

  public Cosmetics(String description, int qty, double price) {
     super(description, qty, price);
  }

  public boolean isTaxed() {
     return isTaxed;
  }

  public boolean isImported() {
     return isImported;
  }

  public void setImported(boolean b) {
     isImported = b;
  }
}
```

Alternative solution: Alternatively, **instead of using inheritance, we can use object composition** to achieve code reuse as discussed in **Q8** in Java section. If you were to use object composition instead of inheritance, you would have classes *Book*, *CD* and *Cosmetics* implementing the *Item* interface directly (*Goods* class would not be required), and make use of a ***GoodsHelper*** class to achieve code reuse through composition.



Let's define a *Tax* class, which is responsible for calculating the tax. The *Tax* class is composed in your *Goods* class, which makes use of **object composition to achieve code reuse**.

```
public class Tax {
```

```
  //stay away from hard coding values. Define constants or read from a ".properties" file
  public static final double SALES_TAX = 0.10; //10%
  public static final double IMPORT_TAX = 0.05; //5%

  private double salesTax = 0.0;
  private double importTax = 0.0;

  public void calculate(boolean isTaxable, boolean isImported, double price) {
    if (isTaxable) {
       salesTax = price * SALES_TAX;
    }
    if (isImported) {
        importTax = price * IMPORT_TAX;
    }
 }

  public double getTotalUnitTax() {
     return this.salesTax + this.importTax;
  }
}
```

Factory method pattern: To create the items shown above we could use the **factory method pattern** as described in **Q46** in Java section. We would also implement the factory class as a singleton using the **singleton design pattern** as described in **Q45** in Java section. The factory method design pattern instantiates a class in a more flexible way than directly calling the constructor. It loosely couples your calling code from the Items it creates like *CD*, *Book*, etc. Let's look at why factory method pattern is more flexible:

- Sometimes factory methods have to return a single instance of a class instead of creating new objects each time or return an instance from a pool of objects.

- Factory methods have to return a subtype of the type requested. It also can request the caller to refer to the returned object by **its interface rather than by its implementation**, which enables objects to be created without making their implementation classes public.

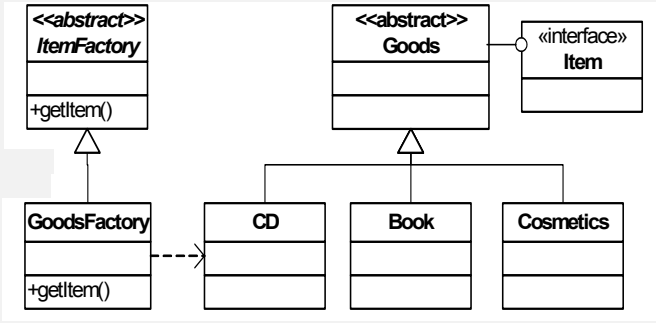
- Sometimes old ways of creating objects can be replaced by new ways of creating the same objects or new classes can be added using polymorphism without changing any of the existing code which uses these objects. **For example:** Say you have a *Fruit* abstract class with *Mango* and *Orange* as its concrete subclasses, later on you can add an *Apple* subclass without breaking the code which uses these objects.

The factory method patterns consist of a **product class hierarchy** and a **creator class hierarchy**.

```
/**
 * ItemFactory is responsible for creating Item objects like CD, Book, and Cosmetics etc
 */
public abstract class ItemFactory {
    public abstract Item getItem(int itemType, String description, int qty, double price) throws ItemException;
}
```

```
/**
 * GoodsFactory responsible for creating Item objects like CD, Book, and Cosmetics etc
 */
public class GoodsFactory extends ItemFactory {

  protected GoodsFactory() { }  //protected so that only ItemFactorySelector within this package can instantiate it to provide a single
                                // point of access (i.e singleton).

  /**
   * Factory method, which decides how to create Items.
   *
   * Benefits are: -- loosely-couples the client (i.e ShoppingBasketBuilder class) from Items such as CD, Book, and Cosmetics etc.
   * In future if we need to create a Book item, which is imported, we can easily incorporate this by adding a new
   * item.TYPE_BOOK_IMPORTED and subsequently adding following piece of code as shown:
   *
   * else if(itemType == TYPE_BOOK_IMPORTED){
   *      item = new Book(description, qty,price);
   *      item.setIsImported(true);
   * }
   *
   * --It is also possible to create an object cache or object pool of our items instead of creating a new instance
```

```
 *    every time without making any changes to the calling class.
 *
 * --Java does not support overloaded constructors which take same parameter list. Instead, use several factory methods.
 *    E.g. getImportedItem(int itemType, String description, int qty, double price).
 */
public Item getItem(int itemType, String description, int qty, double price) throws ItemException {
   Item item = null;
   if (itemType == Item.TYPE_BOOK) {
       item = new Book(description, qty, price);
   } else if (itemType == Item.TYPE_CD) {
       item = new CD(description, qty, price);
   } else if (itemType == Item.TYPE_CD_IMPORTED) {
       item = new CD(description, qty, price);
       item.setImported(true);
   } else if (itemType == Item.TYPE_COSMETICS) {
       item = new Cosmetics(description, qty, price);
   } else {
       throw new ItemException("Invalid ItemType=" + itemType);
   }
   return item;
 }
}
```



Let's use the abstract factory pattern to create an ItemFactory and the singleton pattern to provide a single point of access to the *ItemFactory* returned.

Abstract factory pattern: This pattern is one level of abstraction higher than the factory method pattern because you have an abstract factory (or factory interface) and have multiple concrete factories. Abstract factory pattern usually has a specific method for each concrete type being returned (e.g. createCircle(), createSquare() etc). Alternatively you can have a single method e.g. createShape(…).

Singleton pattern: Ensures that a class has only one instance and provides a global point of access to it (Refer **Q45** in Java section). E.g. a *DataSource* should have only a single instance where it will supply multiple connections from its single *DataSource* pool.

```
/**
 * Abstract factory class which creates a singleton ItemFactory dynamically based on factory name supplied.
 * Benefits of singleton: -- single instance of the ItemFactory -- single point of access (global access within the JVM and the class loader)
 */
public class ItemFactorySelector {
  private static ItemFactory objectFactorySingleInstance = null;
  private static final String FACTORY_NAME = "com.item.GoodsFactory";

  public static ItemFactory getItemFactory() {
    try {
        if (objectFactorySingleInstance == null) {
            //Dynamically instantiate factory and factory name can also be read from a properties file.
            //in future if we need a CachedGoodsFactory which caches Items to improve memory usage then
            //we can modify the FACTORY_NAME to "com.item.CachedGoodsFactory" or conditionally select one of many factories.
           Class klassFactory = Class.forName(FACTORY_NAME);
           objectFactorySingleInstance = (ItemFactory) klassFactory.newInstance();
        }
   }
   catch (ClassNotFoundException cnf) {
       throw new RuntimeException("Cannot create the ItemFactory: " + cnf.getMessage());
   }catch (IllegalAccessException iae) {
       throw new RuntimeException("Cannot create the ItemFactory: " + iae.getMessage());
   }catch (InstantiationException ie) {
       throw new RuntimeException("Cannot create the ItemFactory: " + ie.getMessage());
   }
   return objectFactorySingleInstance;
  }
}
```

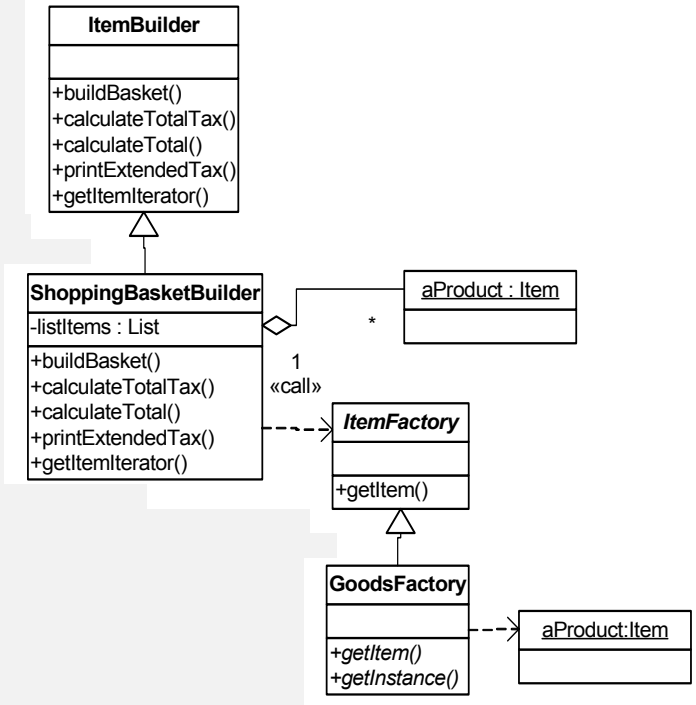
Now we should build a more complex shopping basket object step-by-step, which is responsible for building a basket with items like *CD*, *Book* etc and calculating total tax for the items in the basket. The **builder design pattern** is used to define the interface *ItemBuilder* and the concrete class, which implements this interface, is named *ShoppingBasketBuilder*.

Builder pattern: The subtle difference between the builder pattern and the factory pattern is that in builder pattern, the user is given the **choice to create the type of object he/she wants but the construction process is the same**. But

with the factory method pattern the **factory decides how to create one of several possible classes based on data provided to it**.

```
…//package &  import statements

public interface ItemBuilder {
    public void buildBasket(int itemType, String description, int qty, double unit_price) throws ItemException;
    public double calculateTotalTax() throws ItemException;
    public double calculateTotal() throws ItemException;
    public void printExtendedTaxedPrice() throws ItemException;
    public Iterator getIterator();
}
```

```
…//package & import statements

/**
 * Builder pattern: To simplify complex object creation by defining a class whose purpose is to build instances of another class.
 * There is a subtle difference between a builder pattern and the factory pattern is that in builder pattern, the user is given
 * the choice to create the type of object he/she wants but the construction process is the same. But with the factory method
 * pattern the factory decides how to create one of several possible classes based on data provided to it.
 */
public class ShoppingBasketBuilder implements ItemBuilder {

 private List listItems = null;

 private void addItem(Item item) {
   if (listItems == null) {
     listItems = new ArrayList(20);
   }
   listItems.add(item);
 }

 /**
  * builds a shopping basket
  */
 public void buildBasket(int itemType, String description, int qty, double unit_price) throws ItemException {
     //get the single instance of GoodsFactory using the singleton pattern
     //no matter how many times you call getInstance() you get access to the same instance.
     ItemFactory factory = ItemFactorySelector.getItemFactory();

     //use factory method pattern to create item objects, based on itemType supplied to it.
     Item item = factory.getItem(itemType, description, qty, unit_price);
     this.addItem(item); //adds the item to the basket
 }

 /**
  * calculates total tax on the items in the built basket
  */
 public double calculateTotalTax() throws ItemException {
   if (listItems == null) {
       throw new ItemException("No items in the basket");
   }
   double totalTax = 0.0;
   Iterator it = listItems.iterator();
   while (it.hasNext()) {
       Item item = (Item) it.next();
       totalTax += item.getExtendedTax();
   }
   return totalTax;
 }

 /**
  * calculates total price on the items in the built basket
  */
 public double calculateTotal() throws ItemException {
   if (listItems == null) {
       throw new ItemException("No items in the basket");
   }
   double total = 0.0;
   Iterator it = listItems.iterator();
   while (it.hasNext()) {
       Item item = (Item) it.next();
       total += item.getExtendedTaxPrice();
```



```
    }
    return total;
  }

  /**
   * prints individual prices of the items in the built basket
   */
  public void printExtendedTaxedPrice() throws ItemException {
    if (listItems == null) {
        throw new ItemException("No items in the basket");
    }
    double totalTax = 0.0;
    Iterator it = listItems.iterator();
    while (it.hasNext()) {
        Item item = (Item) it.next();
        System.out.println(item + "" + item.getExtendedTaxPrice());
    }
  }

  public Iterator getIterator() {
      return listItems.iterator();
  }
}
```

Finally, the calling-code, which makes use of our shopping basket builder pattern to build the shopping basket step-by-step and also calculates the taxes and the grand total for the items in the shopping basket.

```
…//package & import statements

public class Shopping {
  /**
   * Class with main(String[] args) method which initially gets loaded by the
   * class loader. Subsequent classes are loaded as they are referenced in the program.
   */
  public static void main(String[] args) throws ItemException {
      process();
  }

  public static void process() throws ItemException {
      //------creational patterns: singleton, factory method and builder design patterns----------
      System.out.println("----create a shopping basket with items ---");
      //Shopping basket using the builder pattern
      ItemBuilder builder = new ShoppingBasketBuilder();
      //build basket of items using a builder pattern
      builder.buildBasket(Item.TYPE_BOOK, "Book - IT", 1, 12.00);
      builder.buildBasket(Item.TYPE_CD, "CD - JAZZ", 1, 15.00);
      builder.buildBasket(Item.TYPE_COSMETICS, "Cosmetics - Lipstick", 1, 1.0);

      //lets print prices and taxes of this built basket
      double totalTax = builder.calculateTotalTax();
      builder.printExtendedTaxedPrice();
      System.out.println("Sales Taxes: " + totalTax);
      System.out.println("Grand Total:   " + builder.calculateTotal());
      System.out.println("----- After adding an imported CD to the basket ----");

      //Say now customer decides to buy an additional imported CD
      builder.buildBasket(Item.TYPE_CD_IMPORTED, "CD - JAZZ IMPORTED", 1, 15.00);

      //lets print prices and taxes of this built basket with imported CD added
      totalTax = builder.calculateTotalTax();
      builder.printExtendedTaxedPrice();
      System.out.println("Sales Taxes: " + totalTax);
      System.out.println("Grand Total:   " + builder.calculateTotal());
  }
}
```

Running the above code produces an output of:

```
----create a shopping basket with items ---
1 Book - IT : 12.0
1 CD - JAZZ : 16.5
1 Cosmetics - Lipstick : 1.1
Sales Taxes: 1.6
```

```
Grand Total:   29.6

----- After adding an imported CD to the basket ----

1 Book - IT : 12.0
1 CD - JAZZ : 16.5
1 Cosmetics - Lipstick : 1.1
1 CD - JAZZ IMPORTED : 17.25
Sales Taxes: 3.85
Grand Total:   46.85
```

Scenario: The XYZ Retail wants to evaluate a strategy to determine items with description longer than 15 characters because it won't fit in the invoice and items with description starting with "CD" to add piracy warning label.

Solution: You can implement evaluating a strategy to determine items with description longer than 15 characters and description starting with "CD" applying the **strategy design pattern** as shown below:

Strategy pattern: The Strategy pattern lets you build software as a loosely coupled collection of interchangeable parts, in contrast to a monolithic, tightly coupled system. Loose coupling makes your software much more extensible, maintainable, and reusable. The main attribute of this pattern is that each strategy encapsulates algorithms i.e. it is **not data based but algorithm based**. Refer **Q10, Q55** in Java section.

Example: You can draw borders around almost all Swing components, including panels, buttons, lists, and so on. Swing provides numerous border types for its components: bevel, etched, line, titled, and even compound. JComponent class, which acts as the base class for all Swing components by implementing functionality common to all Swing components, draws borders for Swing components, using strategy pattern.

```
public interface CheckStrategy {
   public boolean check(String word);
}
```

```
public class LongerThan15 implements CheckStrategy {
   public static final int LENGTH = 15; //constant

   public boolean check(String description) {
    if (description == null)
        return false;
    else
        return description.length() > LENGTH;
   }
}
```

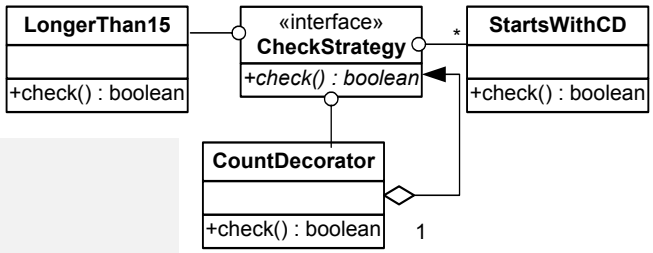
```
public class StartsWithCD implements CheckStrategy {
  public static final String STARTS_WITH = "cd";

  public boolean check(String description) {
     String s = description.toLowerCase();
     if (description == null || description.length() == 0)
        return false;
     else
        return s.startsWith(STARTS_WITH);
  }
}
```

Scenario: The XYZ retail has decided to count the number of items, which satisfy the above strategies.

Solution: You can apply the **decorator design pattern** around your strategy design pattern. Refer **Q20** in Java section for the decorator design pattern used in java.io.*. The decorator acts as a **wrapper** around the *CheckStrategy* objects where by call the check(…) method on the *CheckStrategy* object and if it returns true then increment the counter. The decorator design pattern can be used to provide additional functionality to an object of some kind. The key to a decorator is that a decorator "wraps" the object decorated and looks to a client exactly the same as the object wrapped. This means that the **decorator implements the same interface as the object it decorates**.

Decorator design pattern: You can think of a decorator as a shell around the object decorated. The decorator catches any message that a client sends to the object instead. The decorator may apply some action and then pass the message it received on to the decorated object. That object probably returns a value to the decorator which may again apply an action to that result, finally sending the (perhaps-modified) result to the original client. To the client the **decorator is invisible**. It just sent a message and got a result. However the decorator had two chances to enhance the result returned.

```
public class CountDecorator implements CheckStrategy {

  private CheckStrategy cs = null;
  private int count = 0;

  public CountDecorator(CheckStrategy cs) {
    this.cs = cs;
  }

  public boolean check(String description) {
    boolean isFound = cs.check(description);
    if (isFound)
        this.count++;
    return isFound;
  }

  public int count() {
      return this.count;
  }

  public void reset() {
       this.count = 0;
  }
}
```



There is a subtle difference between the decorator pattern and the proxy pattern is that, the main intent of the decorator pattern is to enhance the functionality of the target object whereas the main intent of the proxy pattern is to control access to the target object.

A decorator object's interface must conform to the interface of the component it decorates

Now, let's see the calling class *Shopping*:

```
//…. package & import statements

public class Shopping {
  //...

  public static void process() throws ItemException {
    ...
    Iterator it = builder.getIterator();
    boolean bol = false;
    CheckStrategy strategy = null;

    it = builder.getIterator();
    //for starting with CD
    strategy = new StartsWithCD();
    strategy = new CountDecorator(strategy);
    while (it.hasNext()) {
        Item item = (Item) it.next();
        bol = strategy.check(item.getDescription());
        System.out.println("\n" + item.getDescription() + " --> " + bol);
    }

    System.out.println("No of descriptions starts with CD -->" + ((CountDecorator) strategy).count());

    it = builder.getIterator();
    //for starting with CD
    strategy = new LongerThan15();
    strategy = new CountDecorator(strategy);
    while (it.hasNext()) {
        Item item = (Item) it.next();
        bol = strategy.check(item.getDescription());
        System.out.println("\n" + item.getDescription() + " --> " + bol);
    }
    System.out.println("No of descriptions longer than 15 characters -->" + ((CountDecorator) strategy).count());
  }
}
```

Running the above code produces an output of:

```
----count item description starting with 'cd' or longer than 15 characters ---
---------- description satarting with cd ----------
Book - IT --> false
CD - JAZZ --> true
Cosmetics - Lipstick --> false
CD - JAZZ IMPORTED --> true
No of descriptions starts with CD -->2
---------- description longer than 15 characters ----------
```

```
Book - IT --> false
CD - JAZZ --> false
Cosmetics - Lipstick --> true
CD - JAZZ IMPORTED --> true
No of descriptions longer than 15 characters -->2
```

Scenario: So far so good, for illustration purpose if you need to adapt the strategy class to the *CountDecorator* class so that you do not have to explicitly cast your strategy classes to *CountDecorator* as shown in bold arrow in the class *Shopping*. We can overcome this by slightly rearranging the classes. The class *CountDecorator* has two additional methods count() and reset(). If you only just add these methods to the interface *CheckStrategy* then the classes *LongerThan15* and *StartsWithCD* should provide an implementation for these two methods. These two methods make no sense in these two classes.

Solution: So, to overcome this you can introduce an adapter class named *CheckStrategyAdapter,* which just provides a bare minimum default implementation.

```
public interface CheckStrategy {
    public boolean check(String word);
    public int count();
    public void reset();
}
```

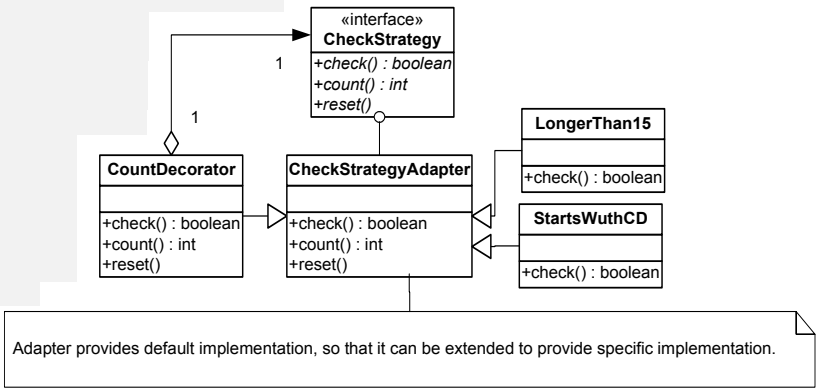
```
/**
 * This is an adapter class which provides default implementations to be extended not to be used and facilitates its subclasses to be
 * adapted to each other. Throws an unchecked exception to indicate improper use.
 */

public class CheckStrategyAdapter implements CheckStrategy {
  public boolean check(String word) {
      throw new RuntimeException("Improper use of CheckStrategyAdapter class method check(String word)" );
 }

 public int count() {
      throw new RuntimeException("Improper use of CheckStrategyAdapter class method count()" );
 }

 public void reset() {
      throw new RuntimeException("Improper use of CheckStrategyAdapter class method reset()" );
 }
}
```

```
public class LongerThan15 extends CheckStrategyAdapter {
 public static final int LENGTH = 15;

 public boolean check(String description) {
    if (description == null)
       return false;
    else
       return description.length() > LENGTH;
  }
}
```

```
public class StartsWithCD extends CheckStrategyAdapter {
 public static final String STARTS_WITH = "cd";

 public boolean check(String description) {
    String s = description.toLowerCase();
    if (description == null || description.length() == 0)
       return false;
   else
       return s.startsWith(STARTS_WITH);
 }
}
```

```
public class CountDecorator extends CheckStrategyAdapter {

 private CheckStrategy cs = null;
 private int count = 0;

 public CountDecorator(CheckStrategy cs) {
     this.cs = cs;
 }
```

```
  public boolean check(String description) {
    boolean isFound = cs.check(description);
    if (isFound)
        this.count++;
    return isFound;
  }

  public int count() {
    return this.count;
  }

  public void reset() {
    this.count = 0;
  }
}
```



Now, let's see the revised calling class *Shopping*:

```
//...package & import statements

public class Shopping {
  //.....
  public static void process() throws ItemException {

    //--------------------------Strategy and decorator design pattern--------------------------------
    System.out.println("----count item description starting with 'cd'or longer than 15 characters ---");
    Iterator it = builder.getIterator();
    boolean bol = false;
    CheckStrategy strategy = null;
    System.out.println("------------------- description satarting with cd ----------------------------");
    it = builder.getIterator();
    //for starting with CD
    strategy = new StartsWithCD();
    strategy = new CountDecorator(strategy);
    while (it.hasNext()) {
        Item item = (Item) it.next();
        bol = strategy.check(item.getDescription());
        System.out.println(item.getDescription() + " --> " + bol);
    }

    System.out.println("No of descriptions starts with CD -->" + strategy.count());

    System.out.println("------------------- description longer than 15 characters ----------------------");
    it = builder.getIterator();
    //for starting with CD
    strategy = new LongerThan15();
    strategy = new CountDecorator(strategy);
    while (it.hasNext()) {
        Item item = (Item) it.next();
        bol = strategy.check(item.getDescription());
        System.out.println(item.getDescription() + " --> " + bol);
    }
    System.out.println("No of descriptions longer than 15 characters -->" + strategy.count());
  }
}
```

The output is:

```
----count item description starting with 'cd'or longer than 15 characters ---
------------------- description satarting with cd ----------------------------
Book - IT --> false
CD - JAZZ --> true
Cosmetics - Lipstick --> false
CD - JAZZ IMPORTED --> true
No of descriptions starts with CD -->2

------------------- description longer than 15 characters ----------------------

Book - IT --> false
CD - JAZZ --> false
Cosmetics - Lipstick --> true
CD - JAZZ IMPORTED --> true
```

```
No of descriptions longer than 15 characters -->2
```

Scenario: The XYZ Retail also requires a piece of code, which performs different operations depending on the type of item. If the item is an instance of *CD* then you call a method to print its catalog number. If the item is an instance of *Cosmetics* then you call a related but different method to print its colour code. If the item is an instance of *Book* then you call a separate method to print its ISBN number. One way of implementing this is using the Java constructs **instanceof** and **explicit type casting** as shown below:

```
it = builder.getIterator();

while(it.hasNext(); ) {
   String name = null;
   Item item = (Item)iter.next();

   if(item instanceof CD) {
        ((CD) item). markWithCatalogNumber();
   } else if (item instanceof Cosmetics) {
        ((Cosmetics) item). markWithColourCode ();
   } else if (item instanceof Book) {
         ((Book) item). markWithISBNNumber();
   }
}
```

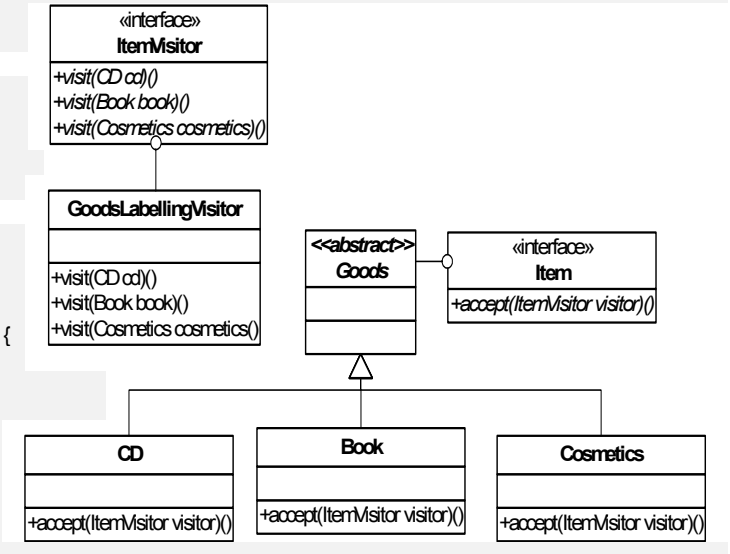
Solution: The manipulation of a collection of polymorphic objects with the constructs **typecasts** and **instanceof** as shown above can get messy and unmaintainable with large elseif constructs and these constructs in frequently accessed methods/ loops can adversely affect performance. You can apply the **visitor** design pattern to avoid using these typecast and "instanceof" constructs as shown below:

Visitor pattern: The visitor pattern makes adding new operations easy and all the related operations are localized in a visitor. The visitor pattern allows you to manipulate a collection of polymorphic objects without the messy and unmaintainable **typecasts** and **instanceof** operations. Visitor pattern allows you to add new operations, which affect a class hierarchy without having to change any of the classes in the hierarchy. **For example** we can add a ***GoodsDebugVisitor*** class to have the visitor just print out some debug information about each item visited etc. In fact you can write any number of visitor classes for the *Goods* hierarchy e.g. ***GoodsLabellingVisitor***, ***GoodsXXXXVisitor etc***.

```
public interface Item {
   //...
   public void accept(ItemVisitor visitor);
}
```

```
public interface ItemVisitor {
    public void visit (CD  cd);
    public void visit (Cosmetics cosmetics);
    public void visit (Book book);
}
```

```
/**
 * visitor class which calls different methods depending
*  on type of item.
 */
public class GoodsLabellingVisitor implements ItemVisitor {

  public void visit(CD cd) {
      markWithCatalogNumber(cd);
  }

  public void visit(Cosmetics cosmetics) {
      markWithColorNumber(cosmetics);
  }

  public void visit(Book book) {
      markWithISBNNumber(book);
  }

  private void markWithCatalogNumber(CD cd) {
     System.out.println("Catalog number for : " + cd.getDescription());
  }

  private void markWithColorNumber(Cosmetics cosmetics) {
```



```
    System.out.println("Color number for : " + cosmetics.getDescription());
  }

  public void markWithISBNNumber(Book book) {
    System.out.println("ISBN number for : " + book.getDescription());
  }
}
```

```
public class CD extends Goods {
  //...
  public void accept(ItemVisitor visitor) {
    visitor.visit(this);
  }
}
```

```
public class Book extends Goods {
  //...
  public void accept(ItemVisitor visitor) {
    visitor.visit(this);
  }
}
```

```
public class Cosmetics extends Goods {
  //...
  public void accept(ItemVisitor visitor) {
    visitor.visit(this);
  }
}
```

Now, let's see the calling code or class *Shopping*:

```
//... package and import statements

public class Shopping {

  public static void process() throws ItemException {

    //visitor pattern example, no messy instanceof and typecast constructs
    it = builder.getIterator();
    ItemVisitor visitor = new GoodsLabellingVisitor ();
    while (it.hasNext()) {
        Item item = (Item) it.next();
        item.accept(visitor);
    }
  }
}
```

The output is:

```
---- markXXXX(): avoid huge if else statements, instanceof & type casts --------
ISBN number for : Book - IT
Catalog number for : CD - JAZZ
Color number for : Cosmetics - Lipstick
Catalog number for : CD - JAZZ IMPORTED
```

Scenario: The XYZ Retail would like to have a functionality to iterate through every second or third item in the basket to randomly collect some statistics on price.

Solution: This can be implemented by applying the **iterator design pattern**.

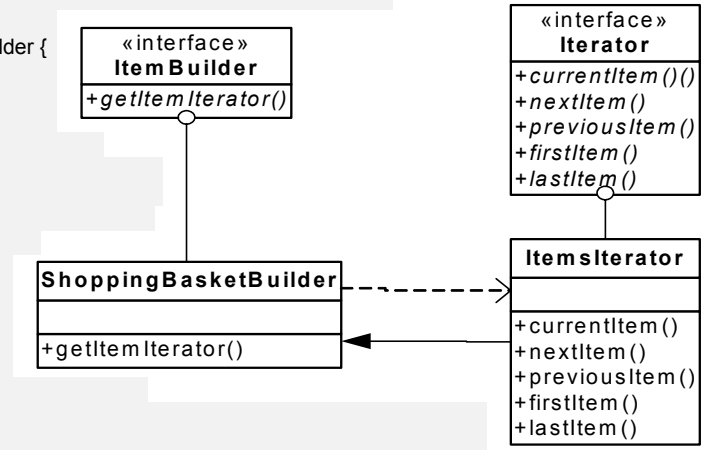
Iterator pattern: Provides a way to access the elements of an aggregate object without exposing its underlying implementation.

```
//… package and import statements

public interface ItemBuilder {
  //..
  public com.item.Iterator getItemIterator();
}
```

```
package com.item;
```

```
public interface Iterator {
    public Item nextItem();
    public Item previousItem();
    public Item currentItem();
    public Item firstItem();
    public Item lastItem();
    public boolean isDone();
    public void setStep(int step);
}
```

```
//… package and import statements

public class ShoppingBasketBuilder implements ItemBuilder {

  private List listItems = null;

  public Iterator getIterator() {
    return listItems.iterator();
  }

  public com.item.Iterator getItemIterator() {
    return new ItemsIterator();
  }

  /**
   * inner class which iterates over basket of items
   */
  class ItemsIterator implements com.item.Iterator {
    private int current = 0;

    private int step = 1;

    public Item nextItem() {
      Item item = null;
      current += step;
      if (!isDone()) {
        item = (Item) listItems.get(current);
      }
      return item;
    }

    public Item previousItem() {
      Item item = null;
      current -= step;
      if (!isDone()) {
        item = (Item) listItems.get(current);
      }
      return item;
    }

    public Item firstItem() {
      current = 0;
      return (Item) listItems.get(current);
    }

    public Item lastItem() {
      current = listItems.size() - 1;
      return (Item) listItems.get(current);
    }

    public boolean isDone() {
      return current >= listItems.size() ? true : false;
    }

    public Item currentItem() {
      if (!isDone()) {
        return (Item) listItems.get(current);
      } else {
        return null;
      }
    }
```



```
    public void setStep(int step) {
      this.step = step;
    }
  }
}
```

Now, let's see the calling code *Shopping*:

```
//… package & import statements

public class Shopping {
  //..

  public static void process() throws ItemException {
    //Iterator pattern example, inner implementations of ShopingBasketBuilder is protected.
    com.item.Iterator itemIterator = builder.getItemIterator();

    //say we want to traverse through every second item in the basket
    itemIterator.setStep(2);
    Item item = null;
    for (item = itemIterator.firstItem(); !itemIterator.isDone(); item = itemIterator.nextItem()) {
        System.out.println("nextItem: " + item.getDescription() + " ====> " + item.getExtendedTaxPrice());
    }

    item = itemIterator.lastItem();
    System.out.println("lastItem: " + item.getDescription() + " ====> " + item.getExtendedTaxPrice());

    item = itemIterator.previousItem();
    System.out.println("previousItem : " + item.getDescription() + "====>" + item.getExtendedTaxPrice());
  }
}
```

The output is:

```
 ---------- steps through every 2nd item in the basket ----------------------
 nextItem: Book - IT ====> 12.0
 nextItem: Cosmetics - Lipstick ====> 1.1
 lastItem: CD - JAZZ IMPORTED ====> 17.25
 previousItem : CD - JAZZ====>16.5
```

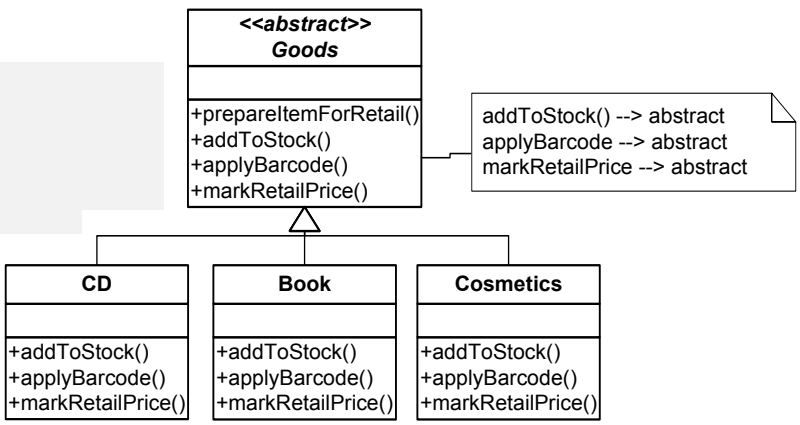
Scenario: The XYZ Retail buys the items in bulk from warehouses and sells them in their retail stores. All the items sold need to be prepared for retail prior to stacking in the shelves for trade. The preparation involves 3 steps for all types of items, i.e. adding the items to stock in the database, applying barcode to each item and finally marking retail price on the item. The preparation process is common involving 3 steps but each of these individual steps is specific to type of item i.e. *Book*, *CD*, and *Cosmetics*.

Solution: The above functionality can be implemented applying the template method design pattern as shown below:

Template method pattern: When you have a sequence of steps to be processed within a method and you want to defer some of the steps to its subclass then you can use a template method pattern. So the template method lets the subclass to redefine some of the steps.

Example Good example of this is the process() method in the Struts *RequestProcessor* class, which executes a sequence of processXXXX(…) methods allowing the subclass to override some of the methods when required. Refer **Q110** in Enterprise section.

```
//...
public abstract class Goods implements Item {
   //...

 /**
  * The template method
  */
 public void prepareItemForRetai() {
    addToStock();
    applyBarcode();
    markRetailPrice();
 }

 public abstract void addToStock();
```



```
  public abstract void applyBarcode();
  public abstract void markRetailPrice();

}
```

```
//..
public class Book extends Goods {
   //..

  //following methods gets called by the template method

  public void addToStock() {
     //database call logic to store the book in stock table.
     System.out.println("Book added to stock : " + this.getDescription());
  }

  public void applyBarcode() {
     //logic to print and apply the barcode to book.
     System.out.println("Bar code applied to book : " + this.getDescription());
  }

  public void markRetailPrice() {
     //logic to read retail price from the book table and apply the retail price.
     System.out.println("Mark retail price for the book : " + this.getDescription());
  }
}
```

```
//...
public class CD extends Goods {
  //..
  //following methods gets called by the template method

  public void addToStock() {
     //database call logic to store the cd in stock table.
     System.out.println("CD added to stock : " + this.getDescription());
  }

  public void applyBarcode() {
     //logic to print and apply the barcode to cd.
     System.out.println("Bar code applied to cd : " + this.getDescription());
  }

  public void markRetailPrice() {
     //logic to read retail price from the cd table and apply the retail price.
     System.out.println("Mark retail price for the cd : " + this.getDescription());
  }
}
```

```
//...
public class Cosmetics extends Goods {
  //...

  public void addToStock() {
     //database call logic to store the cosmetic in stock table.
     System.out.println("Cosmetic added to stock : " + this.getDescription());
  }

  public void applyBarcode() {
     //logic to print and apply the barcode to cosmetic.
     System.out.println("Bar code applied to cosmetic : " + this.getDescription());
  }

  public void markRetailPrice() {
     //logic to read retail price from the cosmetic table and apply the retail price.
     System.out.println("Mark retail price for the cosmetic : " + this.getDescription());
  }
}
```

Now, let's see the calling code *Shopping*:

```
//...
public class Shopping {
  //...
```

```
  public static void process() throws ItemException {
    //...

    Item item = null;
    for (item = itemIterator.firstItem(); !itemIterator.isDone(); item = itemIterator.nextItem()) {
        item.prepareItemForRetai();
        System.out.println("-----------------------------------");
    }
  }
}
```

The output is:

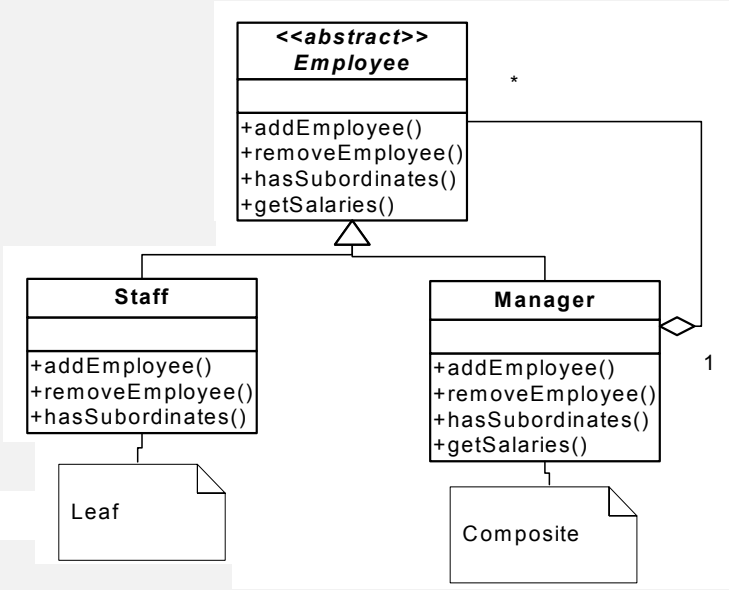
```
 ------------------- prepareItemForRetail() ------------------------------
 Book added to stock : Book - IT
 Bar code applied to book : Book - IT
 Mark retail price for the book : Book - IT
```

Scenario: The employees of XYZ Retail are at various positions. In a hierarchy, the general manager has subordinates, and also the sales manager has subordinates. The retail sales staffs have no subordinates and they report to their immediate manager. The company needs functionality to calculate salary at different levels of the hierarchy.

Solution: You can apply the **composite design pattern** to represent the XYZ Retail company employee hierarchy.

Composite design pattern: The composite design pattern composes objects into tree structures where individual objects like sales staff and composite objects like managers are handled uniformly. Refer **Q52** in Java section or **Q25** in Enterprise section.

```
/**
 * Base employee class
 */
public abstract class Employee {
   private String name;
   private double salary;

  public Employee(String name, double salary) {
      this.name = name;
       this.salary = salary;
  }

  public String getName() {
      return name;
  }

  public double getSalaries() {
      return salary;
  }

  public abstract boolean addEmployee(Employee emp);
  public abstract boolean removeEmployee(Employee emp);
  protected abstract boolean hasSubordinates();
}
```




```
// package & import statements

/**
 * This is the Employee composite class having subordinates.
 */
public class Manager extends Employee {

  List subordinates = null;

  public Manager(String name, double salary) {
     super(name, salary);
  }

  public boolean addEmployee(Employee emp) {
    if (subordinates == null) {
        subordinates = new ArrayList(10);
    }
```

```
    return subordinates.add(emp);
  }

  public boolean removeEmployee(Employee emp) {
    if (subordinates == null) {
        subordinates = new ArrayList(10);
    }
    return subordinates.remove(emp);
  }

  /**
   * Recursive method call to calculate the sum of salary of a manager and his subordinates, which means sum of salary of a manager
   * on whom this method  was invoked and any employees who themselves will have any subordinates and so on.
   */
  public double getSalaries() {
      double sum = super.getSalaries(); //this one's salary

      if (this.hasSubordinates()) {
       for (int i = 0; i < subordinates.size(); i++) {
            sum += ((Employee) subordinates.get(i)).getSalaries();
       }
    }
    return sum;
  }

  public boolean hasSubordinates() {
     boolean hasSubOrdinates = false;
     if (subordinates != null && subordinates.size() > 0) {
          hasSubOrdinates = true;
     }
     return hasSubOrdinates;
  }
}
```

```
/**
 * This is the leaf staff employee object. staff do not have any subordinates.
 */
public class Staff extends Employee {

  public Staff(String name, double salary) {
     super(name, salary);
  }

  public boolean addEmployee(Employee emp) {
     throw new RuntimeException("Improper use of Staff class");
  }

  public boolean removeEmployee(Employee emp) {
      throw new RuntimeException("Improper use of Staff class");
  }

  protected boolean hasSubordinates() {
      return false;
  }
}
```

Now, let's see the calling code *Shopping*:

```
//...
public class Shopping {
  //.....

  public static void process() throws ItemException {
    //....

    System.out.println("---------------------- Employee hierachy & getSalaries() recursively ------------");
    //Employee hierachy

    Employee generalManager = new Manager("John Smith", 100000.00);

    Employee salesManger = new Manager("Peter Rodgers", 80000.00);
    Employee logisticsManger = new Manager("Graham anthony", 90000.00);
```

```
    Employee staffSales1 = new Staff("Lisa john", 40000.00);
    Employee staffSales2 = new Staff("Pamela watson", 50000.00);
    salesManger.addEmployee(staffSales1);
    salesManger.addEmployee(staffSales2);

    Employee logisticsTeamLead = new Manager("Cooma kumar", 70000.00);

    Employee staffLogistics1 = new Staff("Ben Sampson", 60000.00);
    Employee staffLogistics2 = new Staff("Vincent Chou", 20000.00);
    logisticsTeamLead.addEmployee(staffLogistics1);
    logisticsTeamLead.addEmployee(staffLogistics2);

    logisticsManger.addEmployee(logisticsTeamLead);

    generalManager.addEmployee(salesManger);
    generalManager.addEmployee(logisticsManger);

    System.out.println(staffSales1.getName() + "-->" + staffSales1.getSalaries());
    System.out.println(staffSales2.getName() + "-->" + staffSales2.getSalaries());

    System.out.println("Logistics dept " + " --> " + logisticsManger.getSalaries());

    System.out.println("General Manager " + " --> " + generalManager.getSalaries());
  }
}
```

The output is:

```
---------------------- Employee hierachy & getSalaries() recursively -------------
Lisa john-->40000.0
Pamela watson-->50000.0
Logistics dept  --> 240000.0
General Manager  --> 510000.0
```

Scenario: The purchasing staffs (aka logistics staff) of the XYZ Retail Company need to **interact with other subsystems** in order to place purchase orders. They need to communicate with their stock control department to determine the stock levels, also need to communicate with their wholesale supplier to determine availability of stock and finally with their bank to determine availability of sufficient funds to make a purchase.

Solution: You can apply the façade design pattern to implement the above scenario.

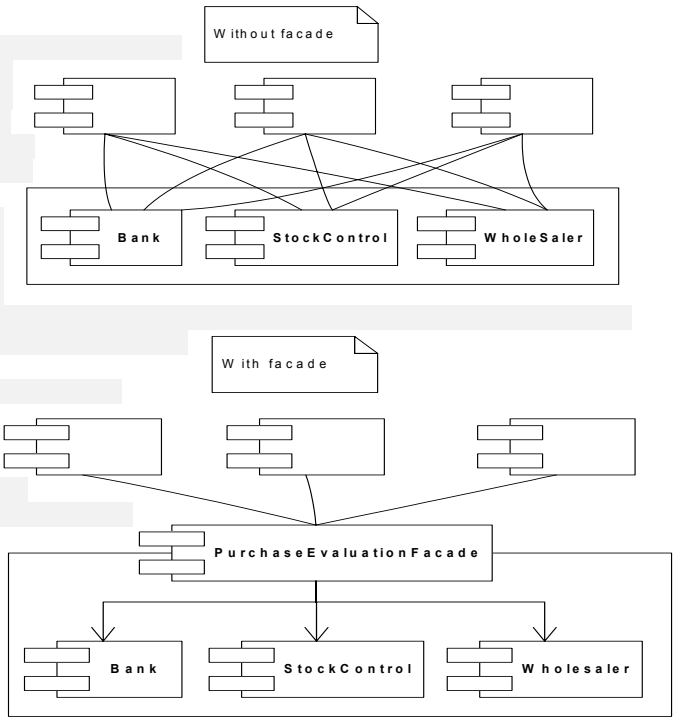
Façade pattern: The façade pattern provides an interface to large subsystems of classes. A common design goal is to minmize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a façade object that provides a single, simplified interface.

```
public class StockControl {
  public boolean isBelowReorderpoint(Item item) {
     //logic to evaluate stock level for item
     return true;
  }
}
```

```
public class Bank {
   public boolean hasSufficientFunds() {
       //logic to evaluate if we have sufficient savings goes here
       return true;
   }
}
```

```
public class WholeSaler {
   public boolean hasSufficientStock(Item item) {
       //logic to evaluate if the wholesaler has enough stock goes here
       return true;
   }
}
```

```
/**
 * This is the facade class
 */
public class PurchaseEvaluation {

  private StockControl stockControl = new StockControl();
```



```
  private WholeSaler wholeSaler = new WholeSaler();
  private Bank bank = new Bank();

  public boolean shouldWePlaceOrder(Item item) {
     if (!stockControl.isBelowReorderpoint(item)) {
         return false;
     }

    if (!wholeSaler.hasSufficientStock(item)) {
         return false;
    }

    if (!bank.hasSufficientFunds()) {
         return false;
    }

    return true;
  }
}
```

Now, let's see the calling code or class *Shopping*:

```
//....

public class Shopping {
  //.......

  public static void process() throws ItemException {

    //....

    //----------------------facade design pattern --------------------------------
    System.out.println("--------------------shouldWePlaceOrder-----------------------------") ;
    PurchaseEvaluation purchaseEval = new PurchaseEvaluation();
    boolean shouldWePlaceOrder = purchaseEval.shouldWePlaceOrder(item);
    System.out.println("shouldWePlaceOrder=" + shouldWePlaceOrder);
  }
}
```

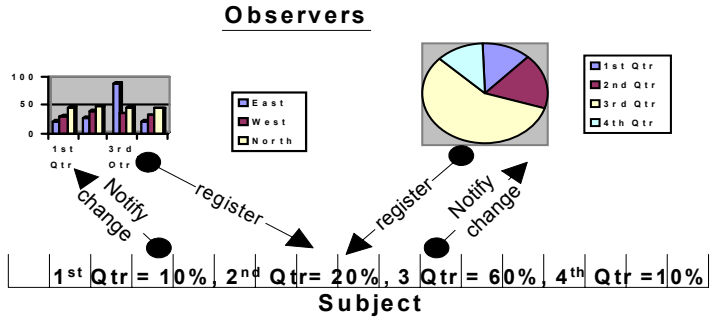
The output is:

```
--------------------shouldWePlaceOrder()----------------------------
shouldWePlaceOrder=true
```

Scenario: The purchasing deprtment also requires functionality where, when the stock control system is updated, all the registered departmental systems like logistics and sales should be notified of the change.

Solution: This can be achieved by applying the observer design pattern as shown below:

Observer pattern: defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. (aka publish-subscribe pattern)



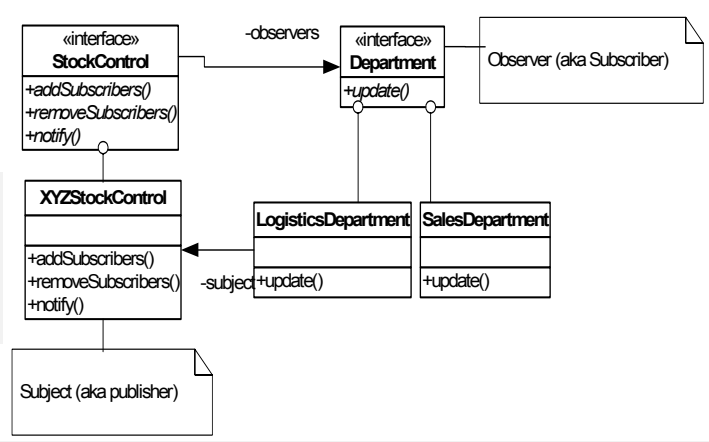

```
/**
 * This is an observer (aka subscriber) interface. This gets notified through its update method.
 */
public interface Department {
    public void update(Item item, int qty);
}
```

```
public class LogisticsDepartment implements Department {
   public void update(Item item, int qty) {
      //logic to update department's stock goes here
      System.out.println("Logistics has updated its stock for " + item.getDescription() + " with qty=" + qty);
   }
}
```

```
public class SalesDepartment implements Department {
  public void update(Item item, int qty) {
    //logic to update department's stock goes here
    System.out.println("Sales has updated its stock for " + item.getDescription() + " with qty=" + qty);
  }
}
```

```
/**
 * Subject (publisher) class: when stock is updated, notifies all the
 * subscribers.
 */
public interface StockControl {
  public void notify(Item item, int qty);
  public void updateStock(Item item, int qty) ;
  public boolean addSubscribers(Department dept);
  public boolean removeSubscribers(Department dept);
}
```

```
//… package & import statements

**
 * publisher (observable) class: when stock is updated
 * notifies all the subscribers.
 */
public class XYZStockControl implements StockControl{

  List listSubscribers = new ArrayList(10);

  //...

  public boolean addSubscribers(Department dept) {
    return listSubscribers.add(dept);
  }

  public boolean removeSubscribers(Department dept) {
    return listSubscribers.remove(dept);
  }

  /**
   * writes updated stock qty into databases
   */
  public void updateStock(Item item, int qty) {
     //logic to update an item's stock goes here
     notify(item, qty); //notify subscribers that with the updated stock info.
  }

  public void notify(Item item, int qty) {
    int noOfsubscribers = listSubscribers.size();
    for (int i = 0; i < noOfsubscribers; i++) {
       Department dept = (Department) listSubscribers.get(i);
       dept.update(item, qty);
    }
  }
}
```



Now, let's see the calling code or class *Shopping*:

```
// package & import statements

public class Shopping {
  //.............
  public static void process() throws ItemException {
    //.....

    //----------------------observer design pattern-----------------------------------------
```

```
    System.out.println("--------------------notify stock update----------------------------");
    Department deptLogistics = new LogisticsDepartment(); //observer/subscriber
    Department salesLogistics = new SalesDepartment(); //observer/subscriber

    StockControl stockControl = new XYZStockControl();//observable/publisher
    //let's register subscribers with the publisher
    stockControl.addSubscribers(deptLogistics);
    stockControl.addSubscribers(salesLogistics);

    //let's update the stock value of the publisher
    for (item = itemIterator.firstItem(); !itemIterator.isDone(); item = itemIterator.nextItem()) {
      if (item instanceof CD) {
        stockControl.updateStock(item, 25);
      } else if (item instanceof Book){
        stockControl.updateStock(item, 40);
      }
      else {
        stockControl.updateStock(item, 50);
      }
    }
  }
}
```

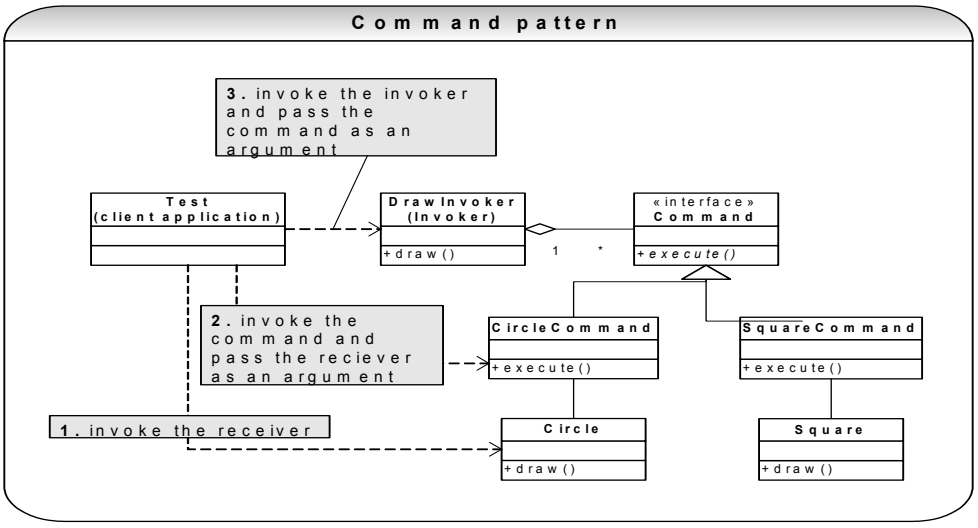
The output is:

```
--------------------notify stock update----------------------------
Logistics has updated its stock for Book - IT with qty=40
Sales has updated its stock for Book - IT with qty=40
Logistics has updated its stock for CD - JAZZ with qty=25
Sales has updated its stock for CD - JAZZ with qty=25
Logistics has updated its stock for Cosmetics - Lipstick with qty=50
Sales has updated its stock for Cosmetics - Lipstick with qty=50
Logistics has updated its stock for CD - JAZZ IMPORTED with qty=25
Sales has updated its stock for CD - JAZZ IMPORTED with qty=25
```

Scenario: The stock control staffs require a simplified calculator, which enable them to add and subtract stock counted and also enable them to undo and redo their operations. This calculator will assist them with faster processing of stock counting operations.

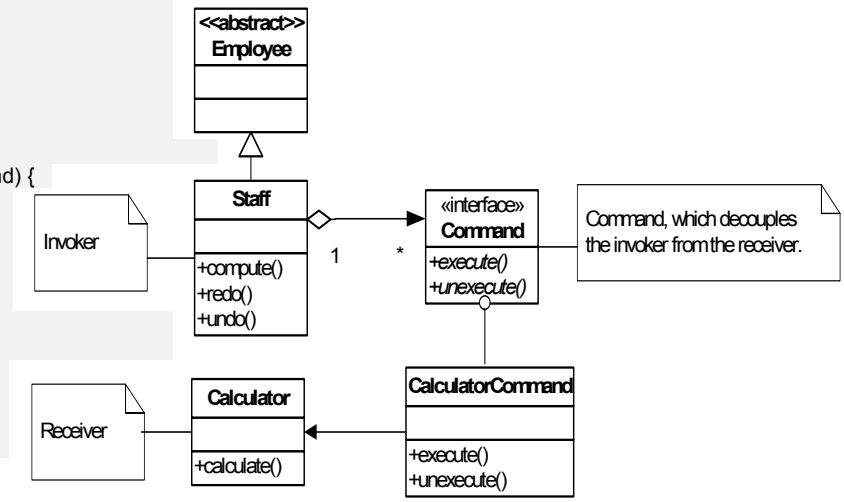
Solution: This can be achieved by applying the **command design pattern** as shown below:

Command pattern: The ***Command*** *pattern* is an object behavioural pattern that allows you to achieve complete decoupling between the sender and the receiver. (A *sender* is an object that invokes an operation, and a *receiver* is an object that receives the request to execute a certain operation. With *decoupling,* the sender has no knowledge of the Receiver's interface.) The term *request* here refers to the command that is to be executed. The Command pattern also allows you to vary when and how a request is fulfilled. At times it is necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. In procedural languages, this type of communication is accomplished via a call-back: a function that is registered somewhere to be called at a later point. Commands are the object-oriented equivalent of call-backs and encapsulate the call-back function.



```
// package & import statements

/**
 * Invoker
 */
public class Staff extends Employee {

  private Calculator calc = new Calculator();
  private List listCommands = new ArrayList(15);
  private int current = 0;

  public Staff(String name) {
    super(name);
  }

  //...
  /**
   * make use of command.
   */
  public void compute(char operator, int operand) {
      Command command = new CalculatorCommand(calc, operator, operand);//initialise the calculator
      command.execute();
      //add commands to the list so that undo operation can be performed
      listCommands.add(command);
      current++;
  }

  /**
   * perform redo operations
   */
  public void redo(int noOfLevels) {
      int noOfCommands = listCommands.size();
      for (int i = 0; i < noOfLevels; i++) {
          if (current < noOfCommands) {
            ((Command) listCommands.get(current++)).execute();
          }
      }
  }

  /**
   * perform undo operations
   */
  public void undo(int noOfLevels) {
    for (int i = 0; i < noOfLevels; i++) {
        if (current > 0) {
            ((Command) listCommands.get(--current)).unexecute();
        }
    }
  }
}
```

```
**
 * actual receiver of the command who performs calculation
 */
public class Calculator {
  private int total = 0;

  /**
   * calculates.
   */
  public void calculate(char operator, int operand) {
    switch (operator) {
    case '+':
        total += operand;
        break;
     case '-':
        total -= operand;
        break;
      }

     System.out.println("Total = " + total);
  }
```



```
}
```

```
/**
 * command interface
 */
public interface Command {
    public void execute();
    public void unexecute();
}
```

```
/**
 * calculator command, which decouples the receiver Calculator from the invoker staff
 */

public class CalculatorCommand implements Command {
   private Calculator calc = null;
   private char operator;
   private int operand;

   public CalculatorCommand(Calculator calc, char operator, int operand) {
       this.calc = calc;
       this.operator = operator;
       this.operand = operand;
  }

  public void execute() {
      calc.calculate(operator, operand);
  }

  public void unexecute() {
      calc.calculate(undoOperand(operator), operand);
  }

  private char undoOperand(char operator) {
    char undoOperator = ' ';
    switch (operator) {
    case '+':
      undoOperator = '-';
      break;

    case '-':
      undoOperator = '+';
      break;
    }
    return undoOperator;
  }
}
```

Now, let's see the calling code or class *Shopping*:

```
//.............
public class Shopping {
  //...........
  public static void process() throws ItemException {

    //---------------------------------------command design pattern-----------------------------------------
    System.out.println("---------------Calculator with redo & undo operations----------------------------");
    Staff stockControlStaff = new Staff("Vincent Chou");

    stockControlStaff.compute('+',10);//10
    stockControlStaff.compute('-',5);//5
    stockControlStaff.compute('+',10);//15
    stockControlStaff.compute('-',2);//13

    //lets try our undo operations
    System.out.println("---------------undo operation : 1 level----------------------------");
    stockControlStaff.undo(1);
    System.out.println("---------------undo operation :2 levels----------------------------");
    stockControlStaff.undo(2);

    //lets try our redo operations
    System.out.println("---------------redo operation : 2 levels----------------------------");
    stockControlStaff.redo(2);
```

```
    System.out.println("----------redo operation : 1 level----------");
    stockControlStaff.redo(1);
  }
}
```

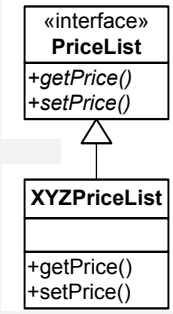
The output is:

```
----------Calculator with redo & undo operations----------
Total = 10
Total = 5
Total = 15
Total = 13
----------undo operation : 1 level----------
Total = 15
----------undo operation :2 levels----------
Total = 5
Total = 10
----------redo operation : 2 levels----------
Total = 5
Total = 15
----------redo operation : 1 level----------
Total = 13
```

Scenario: The XYZ Retail has a 3rd party software component called *XYZPriceList*, which implements an interface PriceList. This 3rd party software component is not thread-safe. So far it performed a decent job since only the sales manager had access to this software component. The XYZ Retail now wants to provide read and write access to all the managers. The source code is not available and only the API is available, so modifying the existing component is not viable. This will cause a dirty read problem if two managers try to concurrently access this component. For example, if the sales manager tries to access an item's price while the logistics manger is modifying the price (say modification takes 1 second), then the sales manager will be reading the wrong value. Let's look at this with a code sample:

```
public interface PriceList {
    public  double getPrice(int itemId) ;
    public void setPrice(int itemId,double newPrice) ;
}
```

```
//…
public class XYZPriceList implements PriceList{

    private static final Map mapPrices = new HashMap(30,.075f);
    public static PriceList singleInstance =  new XYZPriceList();//only one instance

    /**
     * static initializer block
     */
    static {
      //only one item is added to keep it simple
      mapPrices.put(new Integer(1), new Double(12.00));//Book - IT
      //... add more items to price list
    }

    public static PriceList getInstance() {
      return singleInstance;
    }

    public  double getPrice(int itemId) {
      double price = ((Double)mapPrices.get(new Integer(itemId))).doubleValue();
      System.out.println("The price of the itemId " + itemId + " = "+ price);
      return price;
    }

     public  void setPrice(int itemId,double newPrice) {
          System.out.println("wait while mutating price from 12.0 to 15.00 .....");
           try {
              mapPrices.put(new Integer(itemId),new Double(-1));//transient value while updating with a proper value
             Thread.sleep(1000);//assume update/set operation takes 1 second
              mapPrices.put(new Integer(itemId),new Double(newPrice));
           } catch (InterruptedException ie) {}
     }
}
```



The multi-threaded access class:

```
public class PriceListUser implements Runnable {

  int itemId;
  double price;
  static int count = 0;

  public PriceListUser(int itemId) {
    this.itemId = itemId;
  }

  /**
   * runnable code where multi-threads are executed
   */
  public void run() {
    String name = Thread.currentThread().getName();

    if (name.equals("accessor")) {
      price = XYZPriceList.getInstance().getPrice(itemId); //using 3rd party commponent

    } else if (name.equals("mutator")) {
      XYZPriceList.getInstance().setPrice(itemId, 15.00); //using 3rd party commponent
    }
  }
}
```

Now, let's see the calling code or class *Shopping*:

```
//....
public class Shopping {
  //....
  public static void process() throws ItemException {
    //.........

    //-------------------------------------proxy design pattern-----------------------------
    System.out.println("---------------Accessing the price list----------------------------");

    PriceListUser user1 = new PriceListUser(1);//accessing same itemId=1
    PriceListUser user2 = new PriceListUser(1);//accessing same itemId=1

    Thread t1 = new Thread(user1);
    Thread t2 = new Thread(user2);
    Thread t3 = new Thread(user1);

    t1.setName("accessor");//user 1 reads the price
    t2.setName("mutator");//user 2 modifies the price
    t3.setName("accessor");//user 1 reads the price

    t1.start();//accessor user-1 reads before mutator user-2 modifies the price as 12.00
    t2.start();//mutator user-2 sets the price to 15.00
    t3.start();//while the user-2 is setting the price to 15.00 user-1 reads again and gets the price as 12.00
    //user-2 gets the wrong price i.e gets 12.0 again instead of 15.00
  }
}
```

The output is:

```
---------------Accessing the price list----------------------------
The price of the itemId 1 = 12.0
wait while mutating price from 12.0 to 15.00 .....
The price of the itemId 1 = -1.0
```

OR

```
---------------Accessing the price list----------------------------
wait while mutating price from 12.0 to 15.00 .....
The price of the itemId 1 = -1.0
The price of the itemId 1 = -1.0
```

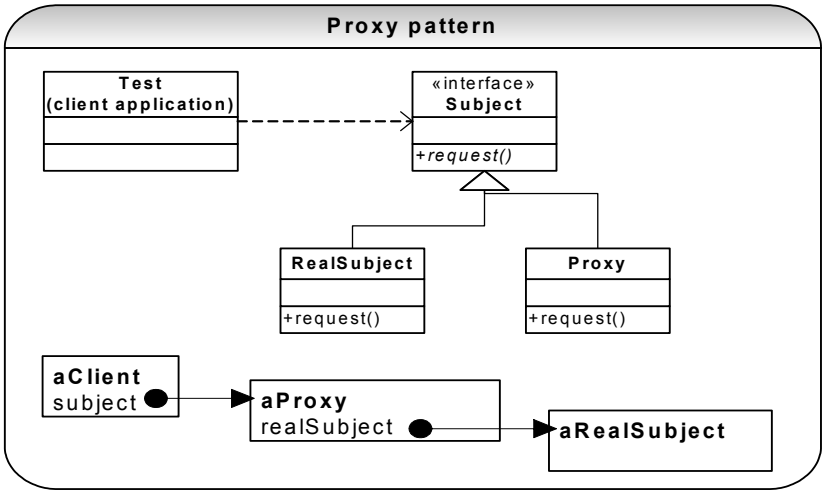
Problem: You get one of the two outputs shown above depending on how the threads initialized by the operating system. The first value of 12.0 is okay and the second value of 12.0 again is a **dirty read** because the value should have been modified to 15.0 by the user-2. So the user-1 reading the value for the second time should get the value of 15.0 after it has been modified.

Solution: This threading issue and inability to modify the existing component can be solved by applying the **proxy design pattern**. You will be writing a proxy class, which will apply the locking for the entries in the *XYZPriceList.* This proxy class internally will be making use of the *XYZPriceList* in a synchronized fashion as shown below:

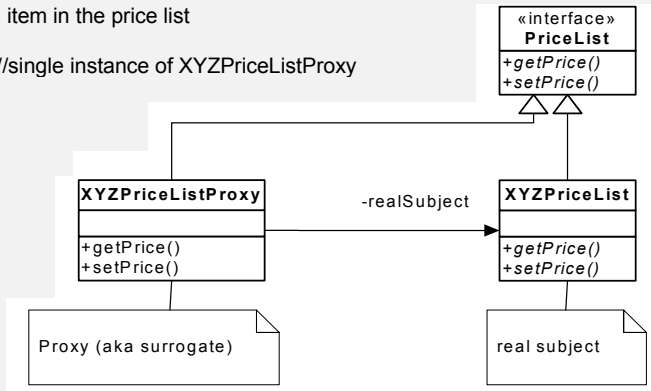
Proxy pattern: Provides a surrogate or placeholder for another object to control access to it. Provide a surrogate or placeholder for another object to control access to it. Proxy object acts as an intermediary between the client and the target object. The proxy object has the same interface as the target object. The proxy object holds reference to the target object. There are different types of proxies:

- **Remote Proxy**: provides a reference to an object, which resides in a separate address space. e.g. EJB, RMI, CORBA etc (RMI stubs acts as a proxy for the skeleton objects.)
- **Virtual Proxy**: Allows the creation of memory intensive objects on demand. The target object will not be created until it is really needed.
- **Access Proxy**: Provides different clients with different access rights to the target object.

Example In Hibernate framework (Refer **Q15 - Q16** in Emerging Technologies/Frameworks section) lazy loading of persistent objects are facilitated by virtual proxy pattern. Say you have a *Department* object, which has a collection of *Employee* objects. Let's say that *Employee* objects are lazy loaded. If you make a call department.getEmployees() then Hibernate will load only the employeeIDs and the version numbers of the *Employee* objects, thus saving loading of individual objects until later. So what you really have is a collection of proxies not the real objects. The reason being, if you have hundreds of employees for a particular department then chances are good that you will only deal with only a few of them. So, why unnecessarily instantiate all the *Employee* objects? This can be a big performance issue in some situations. So when you make a call on a particular employee i.e. employee.getName() then the proxy loads up the real object from the database.




```
/**
 * synchronized proxy class for XYZPriceList
 */
public class XYZPriceListProxy implements PriceList {
  //assume that we only have two items in the pricelist
  Integer[] locks = { new Integer(1), new Integer(2) };//locks for each item in the price list

  public static PriceList singleInstance = new XYZPriceListProxy();//single instance of XYZPriceListProxy

  PriceList realPriceList = XYZPriceList.getInstance(); // real object

  public static PriceList getInstance() {
     return singleInstance;
  }

  public double getPrice(int itemId) {
     synchronized (locks[itemId]) {
        return realPriceList.getPrice(itemId);
     }
  }

  public void setPrice(int itemId, double newPrice) {
```



```
    synchronized (locks[itemId]) {
       realPriceList.setPrice(itemId, newPrice);
    }
  }
}
```

You should make a slight modification to the *PriceListUser* class as shown below in bold.

```
public class PriceListUser implements Runnable {

  int itemId;
  double price;
  static int count = 0;

  public PriceListUser(int itemId) {
     this.itemId = itemId;
  }

  /**
   * runnable code where multi-threads are executed
   */
  public void run() {
     String name = Thread.currentThread().getName();

     if (name.equals("accessor")) {
        price = XYZPriceListProxy.getInstance().getPrice(itemId);

    } else if (name.equals("mutator")) {
        XYZPriceListProxy.getInstance().setPrice(itemId, 15.00);
    }
  }
}
```

Running the same calling code *Shopping* will render the following correct results by preventing dirty reads:

```
----------Accessing the price list----------
The price of the itemId 1 = 12.0
wait while mutating price from 12.0 to 15.00 .....
The price of the itemId 1 = 15.0
```

OR

```
----------Accessing the price list----------
wait while mutating price from 12.0 to 15.00 .....
The price of the itemId 1 = 15.0
The price of the itemId 1 = 15.0
```

What is a dynamic proxy? Dynamic proxies were introduced in J2SE 1.3, and provide an alternate dynamic mechanism for implementing many common design patterns like Façade, Bridge, Decorator, Proxy (remote proxy and virtual proxy), and Adapter. While all of these patterns can be written using ordinary classes instead of dynamic proxies, in many situations dynamic proxies are more compact and can eliminate the need for a lot of handwritten classes. Dynamic proxies are reflection-based and allow you to intercept method calls so that you can interpose additional behaviour between a class caller and its callee. Dynamic proxies are not always appropriate because this code simplification comes at a performance cost due to reflection overhead. Dynamic proxies illustrate the basics of Aspect Oriented Programming (AOP) which complements your Object Oriented Programming. Refer **Q03, Q04** and **Q05** in Emerging Technologies/Frameworks section.

Where can you use dynamic proxies? Dynamic proxies can be used to add crosscutting concerns like logging, performance metrics, memory logging, retry semantics, test stubs, caching etc. Let's look at an example:

InvocationHandler interface is the heart of a proxy mechanism.

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * Handles logging and invocation of target method
 */
public class LoggingHandler implements InvocationHandler {

        protected Object actual;

        public LoggingHandler(Object actual) {
```

```
                this.actual = actual;
        }

        public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {
                try {
                        System.out.println(">>>>>>start executing method: " + method.getName());
                        Object result = method.invoke(actual, args);
                        return result;
                } catch (InvocationTargetException ite) {
                        throw new RuntimeException(ite.getMessage());
                } finally {
                        System.out.println("<<<<<<finished executing method: " + method.getName());
                }
        }
}
```

Let's define the actual interface and the implementation class which adds up two integers.

```
public interface Calculator {
    public int add(int i1, int i2);
}
```

```
public class CalculatorImpl implements Calculator {

        public int add(int i1, int i2) {
                final int sum = i1 + i2;
                System.out.println("Sum is : " + sum);
                return sum;
        }
}
```

Factory method class *CalculatorFactory*, which uses the dynamic proxies when logging, is required.

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

/**
 * singleton factory
 */
public class CalculatorFactory {

    private static CalculatorFactory singleInstance = null;
    private CalculatorFactory() {}

    public static CalculatorFactory getInstance() {
        if (singleInstance == null) {
            singleInstance = new CalculatorFactory();
        }
        return singleInstance;
    }

    public Calculator getCalculator(boolean withLogging) {

        Calculator c = new CalculatorImpl();

        //use dynamic proxy if logging is required, which logs your method calls
        if (withLogging) {
            //invoke the handler, which logs and invokes the target method on the Calculator
            InvocationHandler handler = new LoggingHandler(c);

            //create a proxy
            c = (Calculator) Proxy.newProxyInstance(c.getClass().getClassLoader(), c.getClass().getInterfaces(), handler);
        }

        return c;
    }
}
```

Finally the test class:

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
```

```
public class TestProxy {

   public static void main(String[] args) {
            System.out.println("==============Without dynamic proxy=============");
            Calculator calc = CalculatorFactory.getInstance().getCalculator(false);
            calc.add(3, 2);

            System.out.println("===============With dynamic proxy================");
            calc = CalculatorFactory.getInstance().getCalculator(true);
            calc.add(3, 2);
   }
}
```

The output is:

```
==============Without dynamic proxy=============
Sum is : 5
===============With dynamic proxy================
>>>>>>start executing method: add
Sum is : 5
<<<<<<finished executing method: add
```

Adapter pattern	Sometimes a library cannot be used because its interface is not compatible with the interface required by an application. Also it is possible that we may not have the source code for the library interface. Even if we had the source code, it is not a good idea to change the source code of the library for each domain specific application. This is where you can use an adapter design pattern. Adapter lets classes work together that could not otherwise because of incompatible interfaces. This pattern is also known as a wrapper.
Bridge pattern	Refer **Q41** in Enterprise section.
Chain of responsibility pattern	Refer **Q22** in Enterprise section
Pattern	**Description**
J2EE Patterns	
MVC pattern	Refer **Q3** in Enterprise section Refer **Q54** in Java section
Front controller	Refer **Q24** in Enterprise sectio
Composite View, View Helper, Dispatcher View and Service to Worker	Refer **Q25** in Enterprise section
Business delegate	Refer **Q83** in Enterprise section
Session façade	Refer **Q84** in Enterprise section
Value Object	Refer **Q85** in Enterprise section
Fast lane reader	Refer **Q86** in Enterprise section
Service locator	Refer **Q87** in Enterprise section

Useful links:

- http://www.allapplabs.com/Java_design_patterns/creational_patterns.htm
- http://www.patterndepot.com/put/8/JavaPatterns.htm
- http://www.javaworld.com/columns/jw-Java-design-patterns-index.shtml
- http://www.onjava.com/pub/a/onjava/2002/01/16/patterns.html?page=1
- http://www.corej2eepatterns.com/index.htm
- http://www.theserverside.com/books/wiley/EJBDesignPatterns/index.tss
- http://www.martinfowler.com/eaaCatalog/

Q 12: How would you go about determining the enterprise security requirements for your Java/J2EE application?
A 12: It really pays to understand basic security terminology and J2EE security features. Let's look at them:

Some of the key security concepts are:

- Authentication
- Authorisation ((J2EE declarative & programmatic)
- Data Integrity
- Confidentiality and privacy
- Non-repudiation and auditing

Terminology	Description
Authentication	Authentication is basically an identification process. **Do I know who you are?**

	Terminology used for J2EE security: **Principal:** An entity that can be identified and authenticated. For example an initiator of the request like user etc). **Principal name:** Identity of a principal like user id etc. **Credential:** Information like password or certificate, which can authenticate a principal. **Subject:** A set of principals and their credentials associated with a thread of execution. **Authentication:** The process by which a server verifies the identity presented by a user through username/userid and password or certificate etc. For example the username and password supplied by the user can be validated against an LDAP server or a database server to verify he is whom he claims to be. **Authentication methods:** ❑ **Basic/Digest authentication**: Browser specific and password is encoded using Base-64 encoding. Digest is similar to basic but protects the password through encryption. This is a simple **challenge-response** scheme where the client is challenged for a user id and password. The Internet is divided into ***realms***. A realm is supposed to have one user repository so a combination of user id and password is unique to that realm. The user challenge has the name of the realm so that users with different user ids and password on different systems know which one to apply. Lets look at a HTTP user challenge format WWW-Authenticate: **Basic realm**="*realm_name*" The user-agent (ie Web browser) returns the following HTTP header field Authorization: **Basic *userid:password*** With **Basic authentication** the user id and password string is base64 encoded. The purpose of base64 is to avoid sending possibly unprintable or control characters over an interface that expects text characters. It does not provide any security because the clear text can be readily restored. With **Digest authentication** the server challenges the user with a "nonce" which is an unencrypted random value. The user responds with a **checksum** (typically MD5 hash) of the user id, password, the nonce and some other data. The server creates the same checksum from the user parameters available in the user registry. If both the checksums match then it is assumed that the user knows the correct password. ❑ **Form-based authentication**: Most Web applications use the form-based authentication since it allows applications to customise the authentication interface. Uses base64 encoding which can expose username and password unless all connections are over SSL. (Since this is the most common let us look at in greater detail together ie authentication & authorisation under Authorisation). ❑ **Certificate based authentication**: Uses PKI and SSL. This is by far the most secured authentication method. A user must provide x509 certificate to authenticate with the server.
Authorisation	Authorization is the process by which a program determines whether a given identity is permitted to access a resource such as a file or an application component. **Now that you are authenticated, I know who you are?** But **Are you allowed to access the resource or component you are requesting?** **Terminology used for J2EE security:** **Authorization:** Process of determining what type of access (if any) the security policy allows to a resource by a principal. **Security role:** A logical grouping of users who share a level of access permissions. **Security domain:** A scope that defines where a set of security policies are maintained and enforced. Also known as security policy domain or realm. J2EE uses the concept of **security roles** for both declarative and programmatic access controls. This is different from the traditional model, which is **permission-based** (for example UNIX file system security where resources like files are associated with a user or group who might have permission to read the file but not execute). Let us look at some differences between **Permission-based authorization**: Typically the **permission-based** security both users and resources are defined in a security database and the association of user and groups with the resources takes place through **Access Control Lists** (**ACL**). The maintenance of these registry and ACLs requires a security

administrator.

Role based authorization: In J2EE role based model, the users and groups of users are still stored in a user registry. A mapping must also be provided between **users and groups** to the **security constraints**. This can exist in a registry or **J2EE applications themselves have their own role based security constraints** defined through deployment descriptors like web.xml, ejb-jar.xml, and/or application.xml. So the applications themselves do not have to be defined by a security administrator.

Now lets look at **role based authorization** in a bit more detail:

J2EE has both a **declarative** and **programmatic** way of protecting individual method of each component (Web or EJB) by specifying which security role can execute it.

- Refer **Q23** in Enterprise section.
- Refer **Q81** in Enterprise section
- Also refer **Q7** in Enterprise section for the *deployment descriptors* where **<security-role>** are defined.

Let's look at the commonly used form-based authentication and authorisation in a bit more detail.

STEP:1 The **web.xml** defines the type of authentication mechanism

```
<login-config>
  <auth-method>FORM</auth-method>
    <realm-name>FBA</realm-name>
    <form-login-config>
      <form-login-page>myLogon</form-login-page>
      <form-error-page>myError</form-error-page>
  </form-login-config>
</login-config>
```

STEP: 2 The user creates a form that must contain fields for username, password etc as shown below. The names should be as shown for fields in bold:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="text" name="j_password">
</form>
```

STEP: 3 Set up a security realm to be used by the container. Since LDAP or database provide flexibility and ease of maintenance, Web containers have support for different types of security realms like LDAP, Database etc.

For example Tomcat Web container uses the server.xml to set up the database as the security realm.

```
<realm classname="org.apache.catalina.realm.JDBCRealm" debug="99"
   drivername="org.gjt.mm.mysql.Driver"
   connectionurl="jdbc:mysql://localhost/tomcatusers?user=test;password=test"
   usertable="users" usernamecol="user_name" usercredcol="user_pass"
   userroletable="user_roles" rolenamecol="role_name"/>
```

You have to create necessary tables and columns created in the database.

STEP: 4 Set up the security constraints in the web.xml for authorisation.

```
<security-constraint>
   <web-resource-collection>
     <web-resource-name>PrivateAndSensitive</web-resource-name>
     <url-pattern>/private/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
     <role-name>executive</role-name>
     <role-name>admin</role-name>
   </auth-constraint>
 </security-constraint>
```

The Web containers perform the following steps to implement security when a protected Web resource is accessed:

Step 1: Determine whether the user has been authenticated.

Step 2: If the user has not been already authenticated, request the user to provide security credentials by redirecting the user to the login page defined in the web.xml as per Step-1 & Step-2 described above.

	Step 3: Validate the user credentials against the **security realm** set up for the container. **Step 4**: Determine whether the authenticated user is **authorised** to access the Web resource defined in the deployment descriptor web.xml. Web containers enforce authorization on a page level. For fine grained control programmatic security can be employed using request.getRemoteUser(), request.isUserInRole(), request.getUserPrincipal() etc Note: The Web containers can also propagate the authentication information to EJB containers.
Data integrity	Data integrity helps to make sure if something is **intact and not tampered with** during transmission. **Checksums**: Simply add up the bytes within file or request message. If the checksums match the integrity is maintained. The weakness with this approach is that some times junks can be added to make sums equal like ABCDE == EDCBA **Cryptography hashes:** This uses a mathematical function to create small result called **message digest** from the input message. Difficult to create false positives. Common hash functions are **MD5, SHA** etc. **Data** [e.g. Name is Peter]→**MD5 iterative hash function →Digest** [e.g. f31d120d3] It is not possible to change the message digest back to its original data. You can only compare two message digests i.e. one came with the client's message and the other is recomputed by the server from sent message. If both the message digests are equal then the message is intact and has not been tampered with.
Confidentiality and Privacy	The confidentiality and privacy can be accomplished through encryption. Encryption can be: **Symmetric or private-key:** This is based on a single key. This requires the sender and the receiver to share the same key. Both must have the key. The sender encrypts his message with a private key and the receiver decrypts the message with his own private key. This system is not suitable for large number of users because it requires a key for every pair of individuals who need to communicate privately. As the number of participant increases then number of private keys required also increases. So a company which wants to talk to 1000 of its customers should have 1000 private keys. Also the private keys need to be transmitted to all the participants, which has the vulnerability to theft. **The advantages of the symmetric encryption are its computational efficiency and its security**. **Asymmetric or public-key infrastructure (PKI):** This is based on a pair of mathematically related keys. One is a public key, which is distributed to all the users, and the other key is a private key, which is kept secretly on the server. So this requires only two keys to talk to 1000 customers. This is also called Asymmetric encryption because **the message encrypted by public key can only be decrypted by the private key** and the message encrypted by the private key can only be decrypted by the public key. In a public key encryption anybody can create a key pair and publish the public key. So we need to verify the owner of the public key is who you think it is. So the creator of this false public key can intercept the messages intended for some one else and decrypt it. To protect this public key systems provide mechanisms for validating the public keys using **digital signatures** and **digital certificates**. **Digital signature**: A digital signature is a stamp on the data, which is unique and very difficult to forge. A **digital signature has 2 steps** and establishes 2 things from the security perspective. **STEP 1:** To sign a document means hashing software (e.g. MD5, SHA) will crunch the data into just a few lines by the process called '**hashing**'. These few lines are called **message digest**. It is not possible to change the message digest back to its original data. Same as what we saw above in cryptography hashes**. This establishes whether the message has been modified between the time it was digitally signed and sent and time it was received by the recipient.** **STEP 2:** Computing the digest can verify the integrity of the message but does not stop from some one intercepting it or **verifying the identity of the signer**. This is where encryption comes into picture. Signing the message with the private key will be useful for proving that the message must have come from the user who claims to have signed it. **The second step in creating a digital signature involves encrypting the digest code created in STEP 1 with the sender's private key**. When the message is received by the recipient the following steps take place: 1. Recipient recomputes the digest code for the message. 2. Recipient decrypts the signature by using the sender's public key. This will yield the original digest code of the sender. 3. Compare the original and the recomputed digest codes. If they match then the message is both intact and signed by the user who claims to have signed it (i.e. authentic).

	Digital Certificates: A certificate represents an organisation in an official digital form. This is equivalent to an electronic identity card which serves the purpose of ❑ Identifying the owner of the certificate. This is done with authenticating the owner through trusted 3rd parties called the certificate authorities (CA) e.g. Verisign etc. The CA digitally signs these certificates. When the user presents the certificate the recipient validates it by using the digital signature. ❑ Distributing the owner's public key to his users (or recipients of the message). The **server certificates** let visitors to your website exchange personal information like credit card number etc with the server with the confidence that they are communicating with intended site and not the rogue site impersonating the intended site. Server certificates are must for e-commerce sites. **Personal certificate**s let you authenticate a visitor's identity and restrict access to specified content to particular visitors. Personal certificates are ideal for business-to business communication where offering partners and suppliers special access to your website. A certificate includes details about the owner of the certificate and the issuing CA. A certificate includes: ❑ Distinguished name (DN) of the owner, which is a unique identifier. You need the following for the DN: • Country Name (C) • State (ST) • Locality (L) • Organization Name (O) • Organization Unit (OU) • Common Name (CN) • Email Address. ❑ Public key of the owner. ❑ The issue date of the certificate. ❑ The expiry date of the certificate. ❑ The distinguished name of the issuing CA. ❑ The digital signature of the issuing CA. Now lets look at the core concept of the certificates: **STEP 1:** The owner makes a request to the CA by submitting a certificate request with the above mentioned details. The certificate request can be generated with tool like OpenSSL REQ, Java keytool etc. This creates a certreq.perm file, which can be transferred to CA via FTP. -----**BEGIN NEW CERTIFICATE REQUEST**----- MIIBJTCB0AIBADBtMQswCQYDVQQGEwJVUzEQMA4GA1UEChs4lBMHQXJpem9uYTEN A1UEBxMETWVzYTEfMB0GA1UEChMWTWVs3XbnzYSBDb21tdW5pdHkgQ29sbGVnZTE A1UEAxMTd3d3Lm1jLm1hcmljb3BhLmVkdTBaMA0GCSqGSIb3DQEBAQUAA0kAMEYC QQDRNU6xslWjG41163gArsj/P108sFmjkjzMuUUFYbmtZX4RFxf/U7cZZdMagz4l MmY0F9cdpDLTAutULTsZKDcLAgEDoAAwDQYJKoZIhvcNAQEEBQADQQAjlFpTLgfm BVhc9SQaip5SFNXtzAmhYzvJkt5JJ4X2r7VJYG3J0vauJ5VkjXz9aevJ8dzx37ir 3P4XpZ+NFxK1R= -----**END NEW CERTIFICATE REQUEST**----- **STEP 2:** The CA takes the owner's certificate request and creates a message 'm' from the request and signs the message 'm' with CA's private key to create a separate signature 'sig'. The message 'm' and the signature 'sig' form the certificate, which gets sent to the owner. **STEP 3:** The owner then distributes both parts of the certificate (message and signature) to his customers (or recipients) after signing the certificate with owner's private key. **STEP 4:** The recipient of the certificate (i.e. the customer) extracts the certificate with owner's public key and subsequently verifies the signature 'sig' using CA's public-key. If the signature proves valid, then the recipient accepts the public key in the certificate as the owner's key.
Non-repudiation and auditing	Proof that the sender actually sent the message. It also prohibits the author of the message from falsely denying that he sent the message. This is achieved by record keeping the exact time of the message transmission, the public key used to decrypt the message, and the encrypted message itself. Record keeping can be complicated but critical for non-repudiation.
Secure Socket Layer (SSL)	Secure Socket Layer (SSL) uses a combination of symmetric and asymmetric (public-key) encryption to accomplish confidentiality, integrity, authentication and non-repudiation for Internet communication. In a nutshell SSL uses public key encryption to confidentially transmit a session key which can be used to conduct symmetric encryption. SSL uses the public key technology to negotiate a shared session key between the client and the server. The public key is stored in an X.509 certificate that usually has a digital signature from a trusted 3rd party like Verisign. Lets look at the handshake sequence where the server and the client negotiate the cipher suite to be used, establish a shared session key and authenticate server to

	the client and optionally client to the server. ❑ Client requests a document from a secure server https://www.myapp.com.au). ❑ The server sends its X.509 certificate to the client with its public key stored in the certificate. ❑ The client checks whether the certificate has been issued by a CA it trusts. ❑ The client compares the information in the certificate with the site's public key and domain name. ❑ Client tells the server what cipher suites it has available. ❑ The server picks the strongest mutually available ciphers suite and notifies the client. ❑ The client generates a **session key** (symmetric key or private key**)** and encrypts it using the server's public key and sends it to the server. ❑ The server receives the encrypted session key and decrypts it using its private key. ❑ The client and server use the session key to encrypt and decrypt the data they send to each other.

Note: Use HTTP **post** as opposed to HTTP **get** (sends sensitive information as a query string appended to your URL) in your web based applications, since it is more secured due to hiding sensitive information from your URL query string. Your URL query string can be easily tampered with to determine any security holes in your application.

Q 13: How would you go about describing the open source projects like JUnit (unit testing), Ant (build tool), CVS (version control system) and log4J (logging tool) which are integral part of most Java/J2EE projects?

A 13: JUnit, ANT and CVS are integral part of most Java/J2EE projects. JUnit for unit testing, ANT for deployment, and CVS for source control. Let's look at each, one by one. I will be covering only the key concepts, which can be used as a reference guide in addition to being handy in interviews.

JUnit

This is a regression testing framework, which is used by developers who write unit tests in Java**. Unit testing** is relatively inexpensive and easy way to produce better code faster. Unit testing exercises testing of a very small specific functionality. To run JUnit you should have JUnit.jar in your classpath.

Unix: CLASSPATH=$CLASSPATH:/usr/Java/packages/junit3.8.1/JUnit.jar
Dos: CLASSPATH=%CLASSPATH%;C:\junit3.8.1/JUnit.jar

JUnit can be coded to run in two different modes as shown below:

Per test mode

The per test mode will call the setUp() method before executing every test case and tearDown() method after executing every test case.

Lets look at an example:

```
import junit.framework.TestCase;

public class SampleTest extends TestCase {

  Object o = null;

  protected void setUp() {
     System.out.println("running setUp()");
     //Any database access code
     //Any set up code
     o = new Object();
  }

  protected void tearDown() {
    System.out.println("running tearDown()");
    //Any clean up code
    o = null;
  }

  public void testCustomer() {
     System.out.println("running testCustomer()");
     assertEquals(5, 2 + 3);
     assertNotNull("check if it is null", o);
     assertTrue(5 == 5);
  }

  public void testAccount() {
```

Per suite setup (more common)

In this mode the setUp() and tearDown() will be executed only once:

```
import junit.framework.*;
import junit.extensions.*;


public class SampleTest2 extends TestCase {

Object o = null;

 public void testCustomer() {
    System.out.println("running testCustomer()");
    assertEquals(5, 2 + 3);
 }

 public void testAccount() {
    System.out.println("running testAccount()");
}

 public SampleTest2(String method) {
    super(method);
 }

public static Test suite() {
  TestSuite suite = new TestSuite();

  suite.addTest(new SampleTest2("testCustomer"));
  suite.addTest(new SampleTest2("testAccount"));

  TestSetup wrapper = new TestSetup(suite) {

  protected void setUp() {
```

```
    System.out.println("running testAccount()");
    if (5 < 3)
      fail();
    }

}
```

as per the above example the execution sequence is as follows:

```
running setUp()
running testAccount()
running tearDown()
running setUp()
running testCustomer()
running tearDown()
```

```
      oneTimeSetUp();
 }

 protected void tearDown() {
    oneTimeTearDown();
 }
};

return wrapper;
}

public static void oneTimeSetUp() {
    System.out.println("running setUp()");
    //runs only once to setup
}
public static void oneTimeTearDown() {
    System.out.println("running tearDown()");
   //runs only once to cleanup
}
```

as per the above example the execution sequence is as follows:

```
running setUp()
running testCustomer()
running testAccount()
running tearDown()
```

How to run JUnit?

Text mode: java –cp <junit.jar path> junit.textui.TestRunner
Graphics mode: java –cp <junit.jar path> junit.swingui.TestRunner

The smallest groupings of test expressions are the **methods** that you put them in. Whether you use JUnit or not, you need to put your test expressions into Java methods, so you might as well group the expressions, according to any criteria you want, into methods. An object that you can run with the JUnit infrastructure is a ***Test***. But you can't just implement Test and run that object. You can only *run* specially created instances of **TestCase**. A **TestSuite** is just an object that contains an ordered list of runnable Test objects. TestSuites also implement Test() and are runnable. **TestRunners** execute Tests, TestSuites and TestCases.

ANT (Another Niche Tool)

Ant is a tool which helps you build, test, and deploy (Java or other) applications. ANT is a command-line program that uses a XML file (i.e. build.xml) to describe the build process. The build.xml file describes the various tasks ant has to complete. ANT is a very powerful, portable, flexible and easy to use tool. Ant has the following command syntax:

ant [ant-options] [target 1] [target 2] [….target n]

Some ant options are:

```
-help, -h               : print list of available ant-options (ie prints this message)
-verbose                : be extra verbose
-quiet                  : be extra quiet
-projecthelp , -p       : print project help information
-buildfile  <file>                  : use given build file
-logger <classname>             : class which is to perform logging
-D<property=value> : use value for given property
-propertyfile <name>            : load all properties from file with –D properties taking precedence.
-keep-going, -k                 : execute all targets that do not depend on failed targets
… and more
```

Let’s look at a simple build.xml file:

```
?xml version="1.0" encoding="UTF-8"?>
<project name="MyProject" default="compile" basedir=".">

 <property name="src" value=".\src\" />
```

```
<property name="build" value=".\classes\" />

<target name="init">
 <mkdir dir="${build}" />
</target>

<target name="compile" description="compiles the packages" depends="init">
 <javac srcdir="${src}" destdir="${build}" optimize="on" debug="on">
   <classpath>
            <pathelement location="${build}" />
   </classpath>
  </javac>
</target>

<target name="clean" description="cleans the build directory">
  <delete dir="${build}" />
</target>

</project>
```

We can run the above with one of the following commands

```
$ ant compile
$ ant clean compile
$ ant –b build.xml compile
```

Now lets look at some of the key concepts:

Concept	Explanation with example

Ant Targets	An Ant build file contains one **project**, which itself contains multiple **targets**. Each target contains **tasks**. Targets can depend on each other, so building one target may cause others to be built first.
	
	From the above build.xml file example
	
	name: Name of the target to run.
	
	description: A target determines whether the target defined as internal or public based on description. If the description attribute is defined then it is public and otherwise it is internal. In the above example targets compile and clean are public. The target init is internal. When you run the following command option the public targets are displayed.
	
	`ant -projecthelp`
	
	depends: The target "compile" depends on the target "init". So the target init will be run before target compile is run.
	
	If: If the given property has been defined then the target will be executed.
	
	`<target name="A" if="somePropertyName1">`
	`  <echo message="I am in target A">`
	`</target>`
	
	unless: If the given property is **not defined** then the target will be executed.
	
	`<target name="B" unless="somePropertyName2">`
	`  <echo message="I am in target B">`
	`</target>`
	
	Ant delegates work to other targets as follows:
	
	`<target name="build" depends="prepare">`
	`  <antcall target="compile" />`
	`  <antcall target="jar" />`
	`</target>`

Ant tasks	Ant task is where real work is done. A task can take any number of attributes. Ant tasks can be categorised as follows: ❑ **Core tasks**: Tasks that are shipped with core distribution like <javac …>, <jar …> etc ❑ **Optional tasks**: Tasks that require additional jar files to be executed like <ftp ….> etc ❑ **User defined tasks**: Tasks that are to be developed by users by extending Ant framework.

	For example <javac > is a task.
	
	`<javac srcdir="${src}" destdir="${build}" optimize="on" debug="on">`
	`  <classpath>`
	`  <pathelement location="${build}" />`
	`  </classpath>`
	`</javac>`

Ant data types	Ant data types are different to the ones in other programming languages. Lets look at some of the ant data types.
	
	description:
	
	`<project default=”deploy” basedir=”.”>`
	`  <description> This is my project</description>`
	`</project>`
	
	patternset:
	
	? → matches a single character
	* → matches 0 or more characters
	** → matches 0 or more directory recursively
	
	`<patternset id=”classfile”>`
	`  <include name=”**/*.class” />`
	`  <exclude name=”**/*Test*.class” />`
	`</patternset>`
	
	dirset:
	
	`<dirset dir=”${build.dir}”>`
	`  <patternset id=”classfile”>`
	`  <include name=”**/classes” />`
	`  <exclude name=”**/*Test*” />`
	`  </patternset>`
	`</dirset>`
	
	fileset:
	
	`<fileset dir=”${build.dir}”>`
	`  <include name=”**/*.Java” />`
	`  <exclude name=”**/*Test*” />`
	`</fileset>`
	
	filelist, filemapper,filterchain,filterreader, selectors, xmlcatalogs etc

Lets look at some key tasks where:
 	
Ant updates data from repository.
 	
Carries out unit tests with JUnit.
 	
Builds a jar file if JUnit is success.
 	
Email the results with the help of Ant loggers and listeners.
	Fetch code updates from CVS:
	
	`<target name=”cvsupdate” depends=”prepare”>`
	`  <cvspass cvsroot=”${CVSROOT}” passwd=”${rep.passwd}” />`
	`  <cvs cvsRoot=”${CVSROOT}” command=”update –p -d”  failOnError=”true” />`
	`</target>`
	
	Run unit tests with JUnit:
	
	`<target name=”test” depnds=”compile”>`
	`  <junit failureproperty=”${testsFailed}” >`
	`  <classpath>`
	`  <pathelement path=”${classpath}” />`
	`  <pathelement path=”${build.dir.class}” />`
	`  </classpath>`
	`  <formatter type=”xml”/>`
	`  <test name=”mytests.testall” todir=”${reports.dir}” />`
	`  </junit>`
	`</target>`
	
	Creating a jar file:
	
	`<target name=”jar” depnds=”test” unless=”testsFailed”>`
	`  <jar destfile=”${build.dir}/${name}.jar” basedir=”${build.dir}” include=”**/*.class” />`
	`</target>`

	Email the results with the help of loggers: Now let's look at how we can e-mail the run results. Ant has **listeners** and **loggers**. A listener is a component that is alerted to certain events during the life of a build. A logger is built on top of the listener and is the component that is responsible for logging details about the build. The listeners are alerted to 7 different events like build started, build finished, target started, target finished, task started, task finshed and message logged. The loggers are more useful and interesting. You are always using a logger when you run ant (i.e. DefaultLogger). You can specify the logger as shown below: ant –logger org.apache.tools.ant.listener.**MailLogger** You can also specify other loggers like XmlLogger, Log4Jlistener etc. The MailLogger logs whatever information comes its way and then sends e-mail. A group of properties must be set for a MailLogger which can be passed on to ant as a standard command-line Java option <ie –DmailLogger.mailhost="blah.com" > or the <property …> statements in the init target. Lets look at some of the properties to be set: MailLogger.mailhost MailLogger.from MailLogger.failuire.notify → whether to send an e-mail on build failure. MailLogger.success.notify → whether to send an e-mail on build success. MailLogger.fail.to MailLogger.success.to

Note: Maven is a software project management and comprehension tool, which is gaining popularity. Maven is based on the concept of project object model (**POM**), and it can manage a project's build process, reporting and documentation from a centralized piece of information. Maven provides a uniform build system where by requiring a single set of Ant build files that can be shared by all projects using Maven. Maven provides following information about your project: Change logs from your repository information, cross referenced sources, source metrics, mailing lists, developer lists, dependency lists, unit test reports including coverage etc.

CVS

CVS is a version control or tracking system. It maintains records of files through their development and allows retrieval of any stored version of a file, and supports production of multiple versions.

cvs [cvs-options] command [command-options-arguments]

CVS allows you to split the development into 2 or more parts called a trunk (MAIN) and a branch. You can create 1 or more branches. Typically a branch is used for bug fixes and trunk is used for future development. Both the trunk and branches are stored in the same repository. This allows the change from branch (i.e. bug fixes) to ultimately or periodically be merged into the main trunk ensuring that all bug fixes get rolled into next release.

Unlike some other version control systems, CVS instead of locking files to prevent conflicts (i.e. when 2 developers modifying the same file) it simply allows multiple developers to work on the same file. Subsequently with the aid of cvs file merging feature it allows you to merge all the changes into one file. The benefits of version control systems like CVS include:

- Any stored revision of a file can be retrieved to be viewed or changed.
- Differences between 2 revisions can be displayed.
- Patches can be created automatically.
- Multiple developers can simultaneously work on the same file.
- Project can be branched into multiple streams for varied tasks and then branches can be merged back into trunk (aka **MAIN)**.
- Also supports distributed development and can be configured to record commit messages into a bug tracking system.

Let's look at some of the key concepts and commands.

Concept	Explanation with examples
Building a repository	The repository should be built on a partition that is backed up and won't shut down. The repositories are stored under '**cvsroot**' i.e. /var/lib/cvsroot or /cvsroot. The command to set up the chosen directory as a CVS repository: cvs –d /var/lib/cvsroot

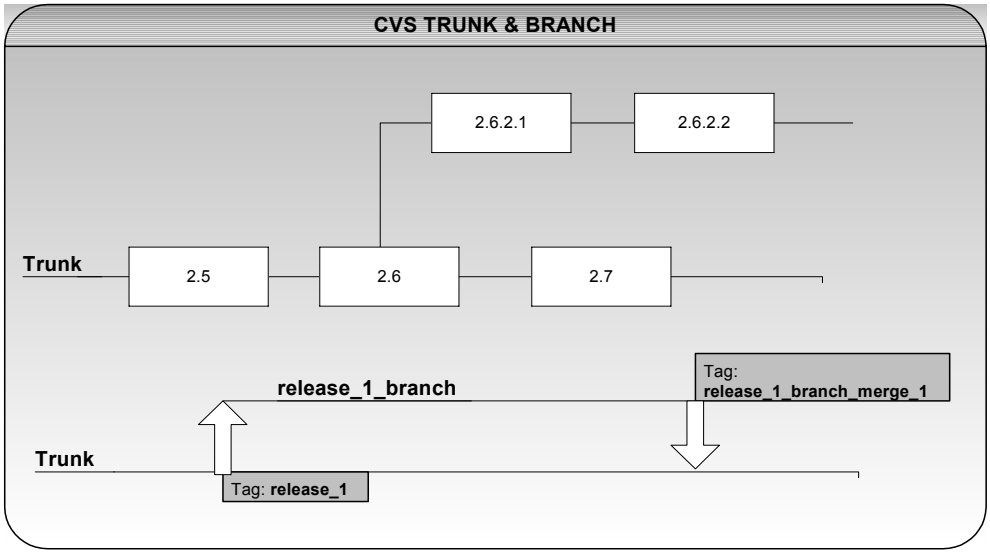
	Lets look at some command line examples:
	
	$ mkdir /var/lib/cvsroot
	$ chgrp team /var/lib/cvsroot
	$ chmod g+srwx /var/lib/cvsroot
	$ **cvs** –d /var/lib/cvsroot

Importing projects	After creating a repository you can import a project or a related collection of files stored under a single directory by using the following command:
	
	cvs [-d <repository-path>] import <project_name> <vendor_tag> <related_tag>
	
	Lets look at some command line examples:
	
	$ cd /tmp
	$ mkdir ProjectX
	$ touch ProjectX/File1.Java
	$ touch ProjectX/File2.Java
	$ touch ProjectX/File3.Java
	$ cd ProjectX
	$ **cvs –d** /var/lib/cvsroot import ProjectX INITIAL start

Creating a **sandbox**, checking out and updating files from cvs repository into a **sandbox**	Copy of the files, which gets checked out by the client from the cvs repository, is called a **sandbox**. The user can manipulate the files within the sandbox and when the files have been modified they can be resubmitted into the repository with the changes. Lets look at how to create a sandbox (i.e. a client working copy): **cvs –d** /var/lib/cvsroot checkout ProjectX The above command will result in creating a subdirectory called ProjectX under the present working directory. Subsequently to keep the sandbox in sync with the repository, an update command can be executed. The update command checks your checked-out cvs sandbox against the cvs repository and down loads any changed files into the sandbox from the repository. **cvs update -d**
Adding files into cvs repository from a sandbox.	To add file from sandbox into cvs repository you should create a file first. $ touch file3 $ **cvs add** file3 $ **cvs commit** To add directories and files $ **cvs add** design plan design/*.rtf plan/*.rtf
Checking file stats and help	$ **cvs** [cvs-options] **stats** [command-option] <filename> $ **cvs –help** $ **cvs rlog** ProjectX

Removing a file from the cvs repository.	To remove a file from the repository, first remove the file from the sandbox directory and then run the cvs command.
	
	$ rm file3
	$ **cvs remove** file3
	$ cvs commit

Moving or renaming files	To move or rename files: $ mv file1 file101 $ **cvs remove** file1 $ **cvs add** file101 $ **cvs commit**
Releasing a sandbox	CVS release should be used before deleting a sandbox. CVS first checks whether there are any files with uncommitted changes.

	$ cvs release
Tagging files	Tagging is a way of marking a group of file revisions as belong together. If you want to look at all the file revisions belonging to a tag the cvs will use the tag string to locate all the files. To tag files in the repository $ **cvs –d** /var/lib/cvsroot **rtag -r** HEAD release_1 ProjectX To tag files in the sandbox $ **cvs tag** release_1
Removing tags	To remove a tag from sandbox. $ **cvs tag –d** release_1 file1 To remove a tag from repository. $ **cvs rtag –d** release_1 file1
Retrieving files based on past revisions instead of the latest files.	We have already looked at how to checkout latest code. What if we want to checkout by a revision? $ **cvs checkout –r** Tagname ProjectX To update by revision $ **cvs update** –d -r release_1
Creating branches	Branches can be added to the repository tree in order to allow different development paths to be tried, or to add parallel development of code to different base versions.  To create a branch from sandbox, you can use $ **cvs update –d -r** release_1 $ **cvs tag –r** release_1 **–b** release_1_branch To create a branch from the repository $ **cvs rtag –r** release_1 **–b** release_1_branch As shown in the diagram it is always a good practice to tag the trunk at the root of branch before branching. This makes it easier to merge the changes back to trunk later. It is also a

	good practice to tag the branch at the root of the branch prior to merging back to head. To merge from branch to trunk (HEAD) cvs update –j branch_base_tag –j branchname $ **cvs update –j** release_1 **–j** release_1_branch To make subsequent merges from the branch to trunk(HEAD) $ **cvs update –j** release_1_branch_merge_1 **–j** release_1_branch To merge from trunk to branch $ **cvs update –j** release_1_branch_merge_1 **–j HEAD**
CVS admin task	To add binary files like images, documents etc to cvs $ **cvs add -kb** image.jpg $ **cvs add -kb** acceptance.doc

Log4J

Refer **Q126** in Enterprise section.

Q 14: How would you go about describing Web services?

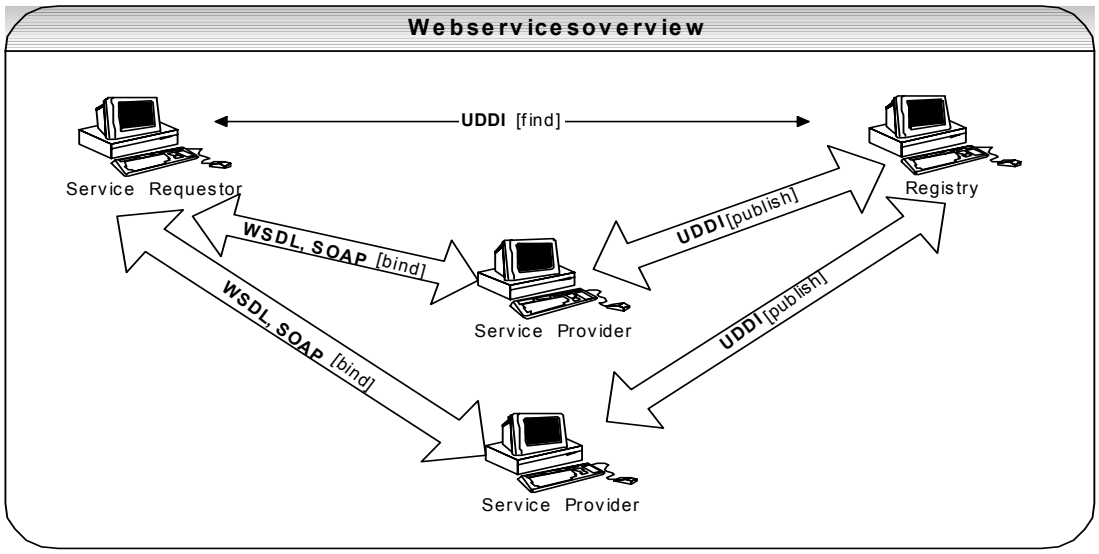
A 14: This book would not be complete without mentioning Web services. Web services provide application-to-application communication using XML.

What is the difference between a Web (website) and a Web service?

Web (website)	Web Service
A **Web** is a scalable information space with interconnected resources. A Web interconnects resources like Web pages, images, an application, word document, e-mail etc.	A **Web service** is a **service,** which lives on the **Web.** A Web service posses both the characteristics of a **Web** and a **service**. We know what a Web is; let's look at what a service is? A service is an application that exposes its functionality through an API (Application Programming Interface). So what is a component you may ask? A **service** is a **component** that can be used remotely through a remote interface either synchronously or asynchronously. The term service also implies something special about the application design, which is called a **service-oriented architecture** (**SOA**). One of the most important features of SOA is the **separation of interface from implementation**. A service exposes its functionality through interface and interface hides the inner workings of the implementation. The client application (ie user of the service) only needs to know how to use the interface. The client does not have to understand actually how the service does its work. For **example**: There are so many different models of cars like MAZDA, HONDA, TOYOTA etc using different types of engines, motors etc but as a user or driver of the car you do not have to be concerned about the internals. You only need to know how to start the car, use the steering wheel etc, which is the interface to you. Usually a service runs on a server, waiting for the client application to call it and ask to do some work? These services are often run on application servers which manage scalability, availability, reliability, multi-threading, transactions, security etc.
Designed to be consumed by humans (ie users, clients, business partners etc). For **example:** www.google.com is a Web search engine that contains index to more than 8 billion of other Web pages. The normal interface is Web browser like Internet Explorer, which is used by human.	Designed to be consumed by software (i.e. other applications). For **example:** Google also provides a Web service interface through the Google API to query their search engine from an application rather than a browser. Refer http://www.google.com/apis/ for Google Web API

Why use Web services when you can use traditional style middleware such as RPC, CORBA, RMI and DCOM?

Traditional middleware	Web Services
Tightly coupled connections to the application and it can break if you make any modification to your application. Tightly coupled applications are hard to maintain and less reusable.	Web Services **support loosely coupled connections**. The interface of the Web service provides a layer of abstraction between the client and the server. The loosely coupled applications reduce the cost of maintenance and increases reusability.
Generally does not support heterogeneity.	Web Services present a new form of middleware based on XML and Web. **Web services are language and platform independent**. You can develop a Web service using any language and deploy it on to any platform, from small device to the largest supercomputer**. Web service uses language neutral protocols such as HTTP and communicates between disparate applications by passing XML messages** to each other via a Web API.
Does not work across Internet.	Does work across Internet.
More expensive and hard to use.	Less expensive and easier to use.



Let's look at some of the key terms

Terms	Explanation
XML	XML provides the way to structure data and XML provides the foundation on which Web services are built.
SOAP	**SOAP** stands for **S**imple **O**bject **A**ccess **P**rotocol. It is an XML based lightweight protocol, which allows software components and application components to communicate, mostly using HTTP. SOAP sits on top of the HTTP protocol. **SOAP is nothing but XML message based document with pre-defined format**. SOAP is designed to communicate via the Internet in a platform and language neutral manner and allows you to get around firewalls as well. Lets look at some SOAP messages: • A SOAP message MUST be encoded using XML • A SOAP message MUST use the SOAP Envelope namespace • A SOAP message MUST use the SOAP Encoding namespace • A SOAP message must NOT contain a DTD reference • A SOAP message must NOT contain XML Processing Instructions <?xml version="1.0"?> <**soap:Envelope** xmlns:soap="http://www.w3.org/2001/12/soap-envelope" soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"> <**soap:Header**> ... ... </**soap:Header**> <**soap:Body**> ... ... <**soap:Fault**>

	... ... </soap:Fault> </soap:Body> </soap:Envelope> Let's look at a SOAP request and a SOAP response: **SOAP Request:** POST /Price HTTP/1.1 Host: www.mysite.com Content-Type: application/soap+xml; charset=utf-8 Content-Length: 300 <?xml version="1.0"?> <soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope" soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"> <soap:Body> <m:GetPrice xmlns:m="http://www.mysite.com/prices"> <m:Item>PlasmaTV</m:Item> </m:GetPrice> </soap:Body> </soap:Envelope> **SOAP Response:** HTTP/1.1 200 OK Content-Type: application/soap; charset=utf-8 Content-Length: 200 <?xml version="1.0"?> <soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope" soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding"> <soap:Body> <m:GetPriceResponse xmlns:m="http://www.mysite.com/prices"> <m:Price>3500.00</m:Price> </m:GetPriceResponse> </soap:Body> </soap:Envelope> Lets look at a HTTP header: **POST** /Price HTTP/1.1 **Host:** www.mysite.com **Content-Type**: text/plain **Content-Length**: 200 **SOAP HTTP Binding** A SOAP method is an HTTP request/response that complies with the SOAP encoding rules. HTTP + XML = SOAP Let's look at a HTTP header containing a soap message: POST /Price HTTP/1.1 Host: www.mysite.com Content-Type: application/soap+xml; charset=utf-8 Content-Length: 200
WSDL (Web Services Description Language)	WSDL stands for **W**eb **S**ervice **D**escription **L**anguage. A WSDL document is an XML document that describes how the messages are exchanged. Let's say we have created a Web service. Who is going to use that and how does the client know which method to invoke and what parameters to pass? There are tools that can generate WSDL from the Web service. Also there are tools that can read a WSDL document and create the necessary code to invoke the Web service. So the WSDL is the Interface Definition Language (IDL) for Web services.
UDDI (Universal Description Discovery	UDDI stands for **U**niversal **D**escription **D**iscovery and **I**ntegration. UDDI provides a way to publish and discover information about Web services. UDDI is like a registry rather than a repository. A registry contains only

and Integration)	reference information like the JNDI, which stores the EJB stub references. UDDI has white pages, yellow pages and green pages. If the retail industry published a UDDI for a price check standard then all the retailers can register their services into this UDDI directory. Shoppers will search the UDDI directory to find the retailer interface. Once the interface is found then the shoppers can communicate with the services immediately. The Web services can be registered for public use at http://www.uddi.org. Once the Web service is selected through the UDDI then it can be located using the discovery process. Before UDDI, there was no Internet standard for businesses to reach their customers and partners with information about their products and services. Neitherr was there a method of how to integrate businesses into each other's systems and processes. UDDI uses WSDL to describe interfaces to Web services

Next section covers some of the popular emerging technologies & frameworks. Some organizations might be considering or already started using these technologies. All these have emerged over the past 3 years. So it is vital that you have at least a basic understanding of these new paradigms and frameworks because these new paradigms and frameworks can offer great benefits such as ease of maintenance, reduction in code size, elimination of duplication of code, ease of unit testing, loose coupling among components, light weight and fine grained objects etc.

SECTION FOUR

Emerging Technologies/Frameworks…

This section covers some of the popular emerging technologies you need to be at least aware of, if you have not already used them. If there are two or more interview candidates with similar skills and experience then awareness or experience with some of the emerging technologies can play a role in the decision. Some organizations might be considering or already started using these technologies. So it is well worth your effort to demonstrate that you understand the basics of technologies like:

- Test Driven Development (**TDD**).
- Aspect Oriented Programming (**AOP**).
- Inversion of Control (**IOC**) (Also known as **Dependency Injection**).
- Annotation or attribute based programming (xdoclet etc).
- Spring framework.
- Hibernate framework.
- EJB 3.0
- JavaServer Faces (**JSF**)

Why should you seriously consider these technologies?

This section covers some of the new and popular design paradigms such as Plain Old Java Objects (POJOs) and Plain Old Java Interfaces (POJI) based services and interceptors, Aspect Oriented Programming (AOP), Dependency Injection (DI), and tools and frameworks, which apply these new paradigms such as Spring, Hibernate, EJB 3.0, XDoclet, JSF, etc. All these have emerged over the past 3 years. These new paradigms and frameworks can offer great benefits such as ease of maintenance, reduction in code size, elimination of duplication of code, ease of unit testing, loose coupling among components, light weight and fine grained objects etc.

Q 01: What is Test Driven Development (TDD)?

A 01: TDD is an iterative software development process where you **first write the test with the idea that it must fail**. This is a different approach to the traditional development where you write the application functionality first and then write test cases. The major benefit of this approach is that the code becomes thoroughly unit tested (you can use **JUnit** or other unit testing frameworks). For JUnit refer **Q13** on "How would you go about…" section. TDD is based on two important principles preached by its originator Kent Beck:

- **Write new business code only if an automated unit test has failed:** Business application requirements drive the tests and tests drive the actual functional code. Each test should test only one business concept, which means avoid writing a single test which tests withdrawing money from an account and depositing money into an account. Any change in the business requirements will impact pre and post conditions of the test. Talking about pre and post conditions, following **design by contract** methodology (Refer **Q9** in Java section) helps achieving TDD. In design by contract, you specify the pre and post conditions that act as contracts of a method, which provides specification to write your tests against.

- **Eliminate duplication from the code:** A particular business concept should be implemented only once within the application code. A code for checking an account balance should be centralized to only one place within the application code. This makes your code decoupled, more maintainable and reusable.

I can hear some of you all saying how can we write the unit test code without the actual application code. Let's look at how it works in steps. The following steps are applied iteratively for business requirements.

STEP: 1 write some tests for a specific business requirement.

STEP: 2 write some basic structural code so that your **test compiles but the test should fail** (failures are the pillars of success). **For example** just create the necessary classes and corresponding methods with skeletal code.

STEP: 3 write the required business code to pass the tests which you wrote in step 1.

STEP: 4 finally refactor the code you just wrote to make it is as simple as it can be. You can refactor your code with confidence that if it breaks the business logic then you have unit test cases that can quickly detect it.

STEP: 5 run your tests to make sure that your refactored code still passes the tests.

STEP: 6 Repeat steps 1-5 for another business requirement.

To write tests efficiently some basic guidelines need to be followed:

- You should be able to run each test in isolation and in any order.
- The test code should not have any duplicate business logic.
- You should test for all the pre and post conditions as well as exceptions.
- Each test should concentrate on one business requirement as mentioned earlier.
- There are many ways to write test conditions so proper care and attention should be taken. In some cases pair programming can help by allowing two brains to work in collaboration. You should have strategies to overcome issues around state of data in RDBMS (Should you persist sample test data, which is a snapshot of your actual data prior to running your tests? Or should you hard code data? Or Should you combine both strategies? Etc).

Q 02: What is the point of Test Driven Development (TDD)?

A 02: TDD process improves your confidence in the delivered code for the following reasons.

- TDD can eliminate duplication of code and also disciplines the developer to focus his mind on delivering what is absolutely necessary. This means the system you develop only does what it is suppose to do

(because you first write test cases for the business requirements and then write the required functionality to satisfy the test cases) and no more.

- These unit tests can be repeatedly run to alert the development team immediately if some one breaks any existing functionality. All the unit tests can be run overnight as part of deployment process and test results can be emailed to the development team.
- TDD ensures that code becomes thoroughly unit tested. It is not possible to write thorough unit tests if you leave it to the end due to deadline pressures, lack of motivation etc.
- TDD complements design by contract methodology and gets the developer thinking in terms of pre and post conditions as well as exceptions.
- When using TDD, tests drive your code and to some extent they assist you in validating your design at an earlier stage.
- TDD also helps you refactor your code with confidence that if it breaks the business logic it gets picked up when you run your unit tests next time.
- TDD promotes **design to interface not implementation** design concept. **For example:** when your code has to take input from an external source or device which is not present at the time of writing your unit tests, you need to create an interface, which takes input from another source in order for your tests to work.

Q 03: What is aspect oriented programming? Explain AOP?

A 03: Aspect-Oriented Programming (**AOP**) complements OOP (Object Oriented Programming) by allowing the developer to dynamically modify the static OO model to create a system that can grow to meet new requirements.

AOP allows you to dynamically modify your static model consisting mainly of business logic to include the code required to fulfil the secondary requirements or in AOP terminology called **cross-cutting concerns** (secondary requirements) like auditing, logging, security, exception handling etc without having to modify the original static model (in fact, we don't even need to have the original code). Better still, we can often keep this additional code in a single location rather than having to scatter it across the existing model, as we would have to if we were using OOP on its own.

For example; A typical Web application will require a servlet to bind the HTTP request to an object and then passes to the business handler object to be processed and finally return the response back to the user. So only a minimum amount of code is initially required. But once you start adding all the other additional secondary requirements or cross-cutting concerns like logging, auditing, security, exception-handling etc the code will inflate to 2-4 times its original size. This is where AOP can assist by separately modularizing these cross-cutting concerns and integrating theses concerns at runtime or compile time through aspect weaving. AOP allows rapid development of evolutionary prototype using OOP by focussing only on the business logic by omitting concerns such as security, auditing, logging etc. Once the prototype is accepted, additional concerns like security, logging, auditing etc can be weaved into the prototype code to transfer it into a production standard application.

AOP nomenclature is different from OOP and can be described as shown below:

Join points: represents the point at which a cross-cutting concern like logging, auditing etc intersects with a main concern like the core business logic. Join points are locations in programs' execution path like method & constructor call, method & constructor execution, field access, class & object initialization, exception handling execution etc.

pointcut: is a language construct that identifies specific join points within the program. A pointcut defines a collection of join points and also provides a context for the join point.

Advice: is an implementation of a cross-cutting concern which is a piece of code that is executed upon reaching a pointcut within a program.

Aspect: encapsulates join points, pointcuts and advice into a reusable module for the cross-cutting concerns which is equivalent to Java classes for the core concerns in OOP. Classes and aspects are independent of one another. **Classes are unaware of the presence of aspects, which is an important AOP concept**. Only pointcut declaration binds classes and aspects.

Weaving is the process for interleaving separate cross-cutting concerns such as logging into core concern such as business logic code to complete the system. AOP weaver composes different implementations of aspects into a cohesive system based on weaving rules. The weaving process (aka injection of aspects into Java classes) can happen at:

- **Compile-time:** Weaving occurs during compilation process.
- **Load-time:** Weaving occurs at the byte-code level at class loading time.
- **Runtime:** Similar to load-time where weaving occurs at byte-code level during runtime as join points are reached in the executing application.

So which approach to use? Load-time and runtime weaving have the advantages of being highly dynamic and enabling changes on the fly without having to rebuild and redeploy. But Load-time and runtime weaving adversely affect system performance. Compile time weaving offers better performance but requires rebuilding and redeployment to effect changes.

Let's look at AOP language constructs with code samples. At a higher level AOP language has two parts:

- **The AOP language specification:** describes language constructs and syntax. The specification has two high level steps:

  STEP 1: Implementation of individual aspects like business logic, logging, security etc into corresponding individual code so that a compiler can translate it into executable code. You can use languages like Java to implement individual code.

  Let's look at a basic code for business logic aspect:

```
public class AccountProcessor {
      public void deposit(Currency amount) {
         //depositing logic only, no log statements
    }

    public void withdraw(Currency amount) throws InsufficientFundsException {
        //withdrawing logic only, no log statements
    }
}
```

  Let's look at a separate basic code for logging aspect

```
public interface Logger {
     public void log (String message);
}
```

  STEP 2: Code weaving rules for individually composed code in step 1 to form a final system. This is achieved through specifying the weaving rules through a language, which is responsible for composing individual code developed in step 1. The weaving rule in our example is:

  - Log public business logic method deposit() at the beginning.

  The language for specifying weaving rules could be an extension of the implementation language. Let's look at a sample specification code for weaving written in *AspectJ* (freely available AOP for Java from Xerox PARC):

```
/** aspect for logging */

public aspect LogAccountProcessor Operations {
   Logger logger = new ConsoleLogger();

   /** join point declaration */
   pointcut logAccount(): call (* AccountProcessor.deposit (..));

   /**  advice to execute before call **/
   before() : logAccount() {

        logger.log("Amount=" + amount);
    }
}
```

- **AOP language implementation:** verifies the code's correctness according to the AOP specification and converts it into byte code that target JVM can execute. AOP language compilers perform two logical steps:

  STEP 1: Combine the individual aspects based on the weaving rules.
  STEP 2: Convert the resulting information into executable byte code.

  For Java based AOP implementation, the JVM would load the weaving rules and then apply those rules to subsequently loaded aspect classes by performing just-in-time aspect weaving.

  The AOP generated resulting code after applying the weaving rules will look something like:

```
public class AccountProcessorWithLogging {
    Logger _logger;

     public void deposit(Currency amount) {
         _logger.log("Amount=" + amount); // joint point defined
         //depositing logic
    }

    public void withdraw(Currency amount) throws InsufficientFundsException {
        //withdrawing logic only, no log statements because no join point defined.
    }
}
```

Q 04: What are the differences between OOP and AOP?
A 04:

Object Oriented Programming (OOP)	Aspect Oriented Programming (AOP)
OOP looks at an application as a set of collaborating objects. OOP code scatters system level code like logging, security etc with the business logic code.	AOP looks at the complex software system as combined implementation of multiple concerns like business logic, data persistence, logging, security, multithread safety, error handling, and so on. Separates business logic code from the system level code. In fact one concern remains unaware of other concerns.
OOP nomenclature has classes, objects, interfaces etc.	AOP nomenclature has join points, point cuts, advice, and aspects.
Provides benefits such as code reuse, flexibility, improved maintainability, modular architecture, reduced development time etc with the help of polymorphism, inheritance and encapsulation.	AOP implementation coexists with the OOP by choosing OOP as the base language. **For example:** AspectJ uses Java as the base language. AOP provides benefits provided by OOP plus some additional benefits which are discussed in the next question.

Q 05: What are the benefits of AOP?
A 05:

- OOP can cause the system level code like logging, transaction, security etc to scatter throughout the business logic. AOP helps overcome this problem by centralising the cross-cutting concerns.

- AOP addresses each aspect separately in a modular fashion with minimal coupling and duplication of code. This modular approach also promotes code reuse by using a business logic concern with a separate logger aspect.

- It is also easier to add newer functionalities by adding new aspects and weaving rules and subsequently regenerating the final code. This ability to add newer functionality as separate aspects enable application designers to delay or defer some design decisions without the dilemma of over designing the application.

- Promotes rapid development of evolutionary prototype using OOP by focussing only on the business logic by omitting cross-cutting concerns such as security, auditing, logging etc. Once the prototype is accepted, additional concerns like security, logging, auditing etc can be weaved into the prototype code to transfer it into a production standard application.

- Developers can concentrate on one aspect at a time rather than having to think simultaneously about business logic, security, logging, performance, multithread safety etc. Different aspects can be developed by different developers based on their key strengths. **For example:** A security aspect can be developed by a security expert or a senior developer who understands security.

Q 06: What is attribute or annotation oriented programming?

A 06: Before looking at attribute oriented programming let's look at code generation processes. There are two kinds of code generation processes.

Passive code generation: is template driven. Input wizards are used in modern IDEs like eclipse, Websphere Studio Application Developer (WSAD) etc where parameters are supplied and the code generator carries out the process of parameter substitution and source code generation. **For example:** in WSAD or eclipse you can create a new class by supplying the "New Java class" wizard appropriate input parameters like class name, package name, modifiers, superclass name, interface name etc to generate the source code. Another example would be *Velocity* template engine, which is a powerful Java based generation tool from the Apache Software Foundation.

Active code generation: Unlike passive code generators the active code generators can inject code directly into the application as and when required.

Attribute/Annotation oriented programming languages leverages the active code generation with the use of declarative tags embedded within the application source code to generate any other kind of source code, configuration files, deployment descriptors etc. These declarative metadata tags are called ***attributes*** or ***annotations***. The purpose of these *attributes* is to extend the functionality of the base language like Java, with the help of custom attributes provided by other providers like Hibernate framework, Spring framework, XDoclet etc. The attributes or annotations are specified with the symbol "**@<label>".** JDK1.5 has a built-in runtime support for attributes.

Let's look at an example. Say we have a container managed entity bean called Account. Using attribute oriented programming we can generate the deployment descriptor file ejb-jar.xml by embedding some attributes within the bean implementation code.

```
/**
 *  @ejb.bean
 *  name="Account"
 *  jndi-name ="ejb/Account"
 */
public abstract class AccountBean implements EntityBean {
    ….
}
```

The above-embedded attributes can generate the ejb-jar.xml as shown below using XDoclet:

```
<ejb-jar>
  <entity>
    <ejb-name>Account</ejb-name>
    <home>com.AccountHome</home>
    <remote>com.Account</remote>
    <ejb-class>com.AccountBean</ejb-class>
  ….

  </entity>
</ejb-jar>
```

Q 07: What are the pros and cons of annotations over XML based deployment descriptors?

A 07: Service related attributes in your application can be configured through a XML based deployment descriptor files or annotations. XML based deployment descriptor files are processed separately from the code, often at runtime, while annotations are compiled with your source code and checked by the compiler.

XML	Annotations
More verbose because has to duplicate a lot of information like class names and method names from your code.	Less verbose since class names and method names are part of your code.
Less robust due to duplication of information which	More robust because annotations are processed with your code

introduces multiple points for failure. If you misspell a method name then the application will fail.	and checked by the compiler for any discrepancies and inaccuracies.

More flexible since processed separately from the code. Since it is not hard-coded can be changed later. Your deployment team has a great flexibility to inspect and modify the configuration.	Less flexible since annotations are embedded in Java comment style within your code.
	
	For example, to define a stateless session EJB 3.0 with annotations, which can serve both local and remote clients:
	
	@Stateless
	@Local ({LocalCounter.class})
	@Remote ({RemoteCounter.class})
	
	public class CounterBean implements LocalCounter, RemoteCounter {
	...
	}

XML files can express complex relationships and hierarchical structures at the expense of being verbose.	Annotations can hold only a small amount of configuration information and most of plumbing has to be done in the framework.

Which one to use? Annotations are suitable for most application needs. XML files are more complex and can be used to address more advanced issues. XML files can be used to override default annotation values. Annotations cannot be used if you do not have access to source-code. The decision to go with annotation or XML depends upon the architecture behind the framework. For example Spring is primarily based on XML and EJB 3.0 is primarily based on annotations, but both support annotations and XML to some degree. EJB 3.0 uses XML configuration files as an optional overriding mechanism and Spring uses annotations to configure some Spring services.

Q 08: What is XDoclet?

A 08: **XDoclet** is an open source code generation engine for **attribute oriented programming** from SourceForge.net (http://xdoclet.sourceforge.net/xdoclet/index.html). So you add attributes (i.e. metadata) in JavaDoc style tags (@ejb.bean) and XDoclet will parse your source files and JavaDoc style attributes provided in the Java comment with @ symbol to generate required artifacts like XML based deployment descriptors, EJB interfaces etc. XDoclet can generate all the artifacts of an EJB component, such as remote & local interfaces as well as deployment descriptors. You place the required attributes on the relevant classes and methods that you want to process.

Q 09: What is inversion of control (**IOC**) (also known as **dependency injection**)?

A 09: Inversion of control or dependency injection is a term used to resolve component dependencies by injecting an instantiated component to satisfy dependency as opposed to explicitly requesting a component. So components will not be explicitly requested but components are provided as needed with the help of an Inversion of controller containers. This is analogous to the Hollywood principal where the servicing components say to the requesting client code “don’t call us, we’ll call you”. Hence it is called inversion of control.

Most of you all are familiar with the software development context where client code (requesting code) collaborates with other dependent components (or servicing components) by knowing which components to talk to, where to locate them and how to talk with them. This is achieved by embedding the code required for locating and instantiating the requested components within the client code. The above approach will tightly couple the dependent components with the client code. This tight coupling can be resolved by applying the **factory design pattern** and **program to interfaces not to implementations driven development**. But the factory design pattern is still an intrusive mechanism because servicing components need to be requested explicitly. Let us look at how dependency injection comes to our rescue. It takes the approach that clients declare their dependency on servicing components through a configuration file (like xml) and some external piece of code assumes the responsibility of locating and instantiating these servicing components and supplying the relevant references when needed to the client code. This external piece of code is often referred to as IOC (aka dependency injection) container or framework.

IOC or dependency injection containers generally control creation of objects (by calling “new”) and resolve dependencies between objects it manages. Spring framework, Pico containers, Hivemind etc are IoC containers to name a few. IOC containers support **eager instantiation**, which is quite useful if you want self-starting services that “come up” on their own when the server starts. They also support **lazy loading**, which is useful when you have many services which may only be sparsely used. Here is pseudo code for how IOC would work:

XML declaration (beans.xml) showing objects that need to be instantiated and dependencies between them. Dependency is declared as property element.

```
<beans>
   <bean id="FlightReservation" class="com.FlightReservation" />
   <bean id="FlightReservation" class="com.HotelReservation" />
   <bean id="TripPlanner" class="com.TripPlanner">
        <property name="flight"><ref bean="FlightReservation"/></property>
        <property name="hotel"><ref bean="HotelReservation"/></property>
  </bean>
  ……
</beans>
```

To initialize the container

```
ClassPathResource res = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(res);
```

The references to the implementation can be retrieved based on "id" attribute in the xml configuration file and all the dependent components are implicitly instantiated in specified order and setter methods (i.e. **setter injection**) are called to resolve the dependencies.

```
factory.getBean("TripPlanner");
```

Dependencies can be wired by either using Java code or using XML.

Q 10: What are the different types of dependency injections?
A 10: There are three types of dependency injections.

- Constructor Injection (e.g. Pico container, Spring etc): Injection is done through constructors.
- Setter Injection (e.g. Spring): Injection is done through setter methods.
- Interface Injection (e.g. Avalon): Injection is done through an interface.

Q 11: What are the benefits of IOC (aka Dependency Injection)?
A 11:

- Minimises the amount of code in your application. With IOC containers you do not care about how services are created and how you get references to the ones you need. You can also easily add additional services by adding a new constructor or a setter method with little or no extra configuration.

- Make your application more testable by not requiring any singletons or JNDI lookup mechanisms in your unit test cases. IOC containers make unit testing and switching implementations very easy by manually allowing you to inject your own objects into the object under test.

- Loose coupling is promoted with minimal effort and least intrusive mechanism. The factory design pattern is more intrusive because components or services need to be requested explicitly whereas in IOC the dependency is injected into requesting piece of code. Also some containers promote the design to interfaces not to implementations design concept by encouraging managed objects to implement a well-defined service interface of your own.

- IOC containers support eager instantiation and lazy loading of services. Containers also provide support for instantiation of managed objects, cyclical dependencies, life cycles management, and dependency resolution between managed objects etc.

Q 12: What is the difference between a service locator pattern and an inversion of control pattern?
A 12:

Service locator	Inversion of Control (IOC)
The calling class which needs the dependent classes needs to tell the service locator which classes are needed. Also the calling classes have the responsibility of finding these dependent classes and invoking them. This makes the classes tightly coupled with each other.	In IoC (aka Dependency Injection) pattern the responsibility is shifted to the IoC containers to locate and load the dependent classes based on the information provided in the descriptor files. Changes can be made to the dependent classes by simply modifying the descriptor files. This approach makes the dependent classes loosely coupled with the calling class.

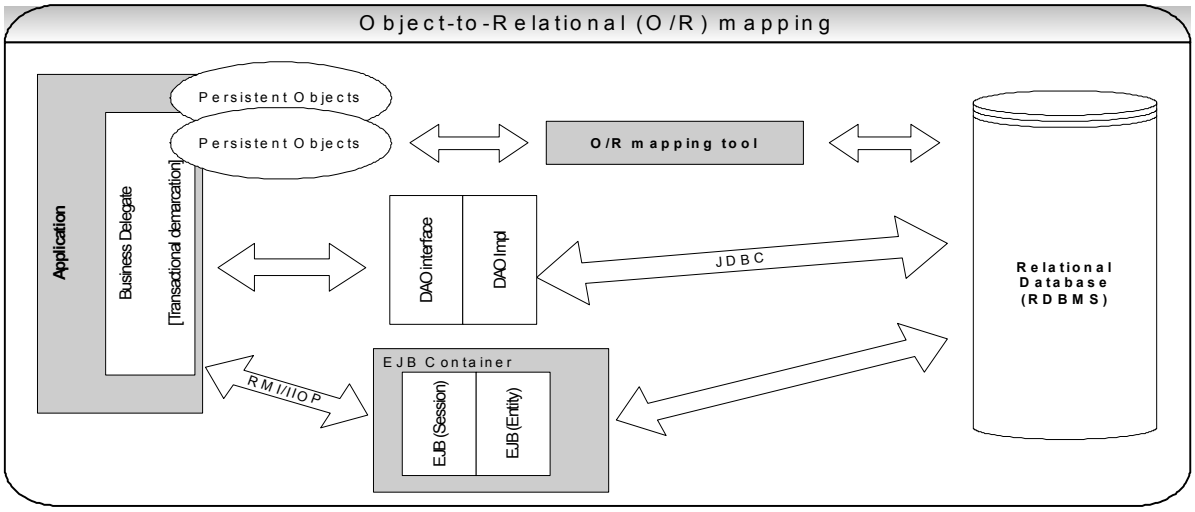
Difficult to unit test the classes separately due to tight coupling.	Easy to unit test the classes separately due to loose coupling.

Q 13: Why dependency injection is more elegant than a JNDI lookup to decouple client and the service?
A 13: Here are a few reasons why a JNDI look up is not elegant:

- The client and the service being looked up must agree on a string based name, which is a contract not enforced by the compiler or any deployment-time checks. You will have to wait till runtime to discover any discrepancies in the string based name between the lookup code and the JNDI registry.
- The JNDI lookup code is verbose with its own try-catch block, which is repeated across the application.
- The retrieved service objects are not type checked at compile-time and could result in casting error at runtime.

Dependency injection is more elegant because it promotes loose coupling with minimal effort and least intrusive mechanism. Dependency is injected into requesting piece of code by the IOC containers like Spring etc. With IOC containers you do not care about how services are created and how you get references to the ones you need. You can also easily add additional services by adding a new constructor or a setter method with little or extra configuration.

Q 14: Explain Object-to-Relational (O/R) mapping?
A 14: There are several ways to persist data and the persistence layer is one of the most important layers in any application development. O/R mapping is a technique of mapping data representation from an object model to a SQL based relational model.



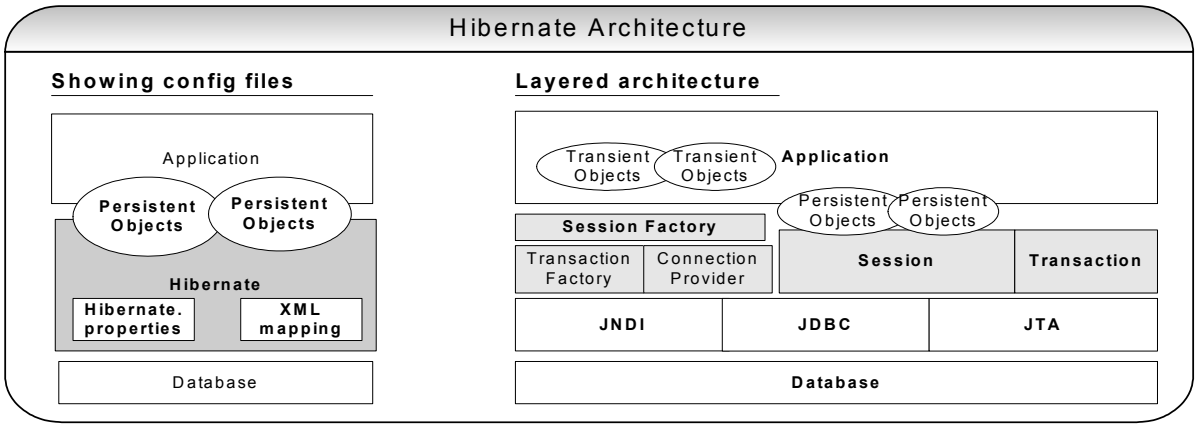
O/R mapping is well suited for read → modify → write centric applications and not suited for write centric applications (i.e. batch processes with large data sets like 5000 rows or more) where data is seldom read. O/R mapping tools allow you to model inheritance (Refer **Q101** in Enterprise section), association and composition class relationships. O/R mapping tools work well in 80-90% of cases and use most of the basic database features like stored procedures, triggers etc, when O/R mapping is not appropriate. Keep in mind that no one size fits all solution. Always validate your architectural design with a vertical slice and test for performance. Some times you have to handcraft your SQL and a good O/R mapping tool should allow that. O/R mapping tools allow your application to be less verbose, more portable and more maintainable.

Q 15: Give an overview of hibernate framework?
A 15: Hibernate is a full-featured, open source Object-to-Relational (O/R) mapping framework. Unlike EJB, Hibernate can work inside or outside of a J2EE container. Hibernate works with Plain Old Java Objects (POJOs), which is much like a JavaBean.

SessionFactory is Hibernate's concept of a single datastore and is threadsafe so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database. A

SessionFactory is usually only built once at startup with a load-on-startup servlet. SessionFactory should be wrapped in some kind of singleton so that it can be easily accessed in an application code.

```
SessionFactory sf = new Configuration().configure().buildSessionfactory();
```



Session is a non-threadsafe object that represents a single unit-of-work with the database. Sessions are opened by a SessionFactory and then are closed when all work is complete. To avoid creating too many sessions ThreadLocal class can be used as shown below to get the current session no matter how many times you make call to the currentSession() method.

```
…
public class HibernateUtil {
   …
  public static final ThreadLocal local = new ThreadLocal();

  public static Session currentSession() throws HibernateException {
            Session session = (Session) local.get();
            //open a new session if this thread has no session
            if(s == null) {
                     session = sf.openSession();
                     local.set(session);
            }

            return session;
  }
}
```

It is also vital that you close your session after your unit of work completes.

Transaction is a single threaded, short lived object used by the application to specify atomicity. Transaction abstracts your application code from underlying JDBC, JTA or CORBA transaction. At times a session can span several transactions. A **TransactionFactory** is a factory for transaction instances. A **ConnectionProvide**r is a factory for pool of JDBC connections. A ConnectionProvider abstracts an application from underlying Datasource or DriverManager.

```
Transaction tx = session.beginTransaction();
Employee emp = new Employee();
emp.setName("Brian");
emp.setSalary(1000.00);

session.save(emp);
tx.commit();
//close session
```

Persistent Objects and Collections are short lived single threaded objects, which store the persistent state and some business functions. They are Plain Old Java Objects (POJOs) and are currently associated with one session. As soon as the associated session is closed, persistent objects are detached and free to use directly as transient data transfer objects in any application layers like business layer, presentation layer etc.

Transient Object and Collections are instances of persistent objects that are currently not associated with a session. They can be either instantiated by an application, which has not yet been persisted or they may have been instantiated by a closed session.

Configuring Hibernate

Hibernate mappings can be either configured manually through a simple readable XML format (*.hbm.xml) or by embedding mapping information directly in the source code using Hibernate doclet (@hibernate.<tags>), which is a sibling to XDoclet and automatically generating the mapping XML file using an Ant script.

Querying with Hibernate

Hibernate provides very robust querying API that supports query strings, named queries and queries built as aggregate expressions. The most flexible way is using the Hibernate Query Language syntax (HQL), which is easy to understand and is an Object Oriented extension to SQL, which supports inheritance and polymorphism.

```
Query query = session.createQuery("Select emp from Employee as emp where emp.sex = :sex");
query.setCharacter("sex",'F');
```

Type-safe queries can be handled by object oriented query by criteria approach. You can also use Hibernate's direct SQL query feature. If none of the above meets your requirements then you can get a plain JDBC connection from a Hibernate session.

Q 16: Explain some of the pitfalls of Hibernate and explain how to avoid them?
A 16:

- Use the ThreadLocal session pattern when obtaining Hibernate session objects (Refer **Q15** in Emerging Technologies/Frameworks). This is important because Hibernate's native API does not use the current thread to maintain associations between session and transaction or between session and application thread.

- Handle resources properly by making sure you properly flush and commit each session object when persisting information and also make sure you release or close the session object when you are finished working with it. Most developers fall into this pitfall. If you pass a connection object to your session object then remember to issue **session.close().close ()** which will first release the connection back to the pool and then will close the session. If you do not pass a connection object then issue **session.close()** to close the session.

- Use **lazy associations** when you use relationships otherwise you can unwittingly fall into the trap of executing unnecessary SQL statements in your Hibernate applications. Let us look at an example: Suppose we have a class *Employee* with many-to-one relationship with class *Department*. So one department can have many employees. Suppose we want to list the name of the employees then we will construct the query as follows:

  ```
  Query query = session.createQuery("from Employee emp");
  List list =  query.list();
  ```

  Hibernate will generate the following SQL query:

  ```
  SELECT <fields> from Employee;
  ```

  If it only generates the query above then it is okay and it serves our purpose, but we get another set of SQL queries without asking it to do anything. One for each of the referenced departments in *Department* table. If you had 5 departments then the following query will be executed 5 times with corresponding department id. This is the N+1 selects problem. In our example it is 5 + 1. Employee table is queried once and Department table is queried 5 times.

  ```
  SELECT <fields> from Department where DEPARTMENT.id=?
  ```

  Solution is to make the *Department* class lazy, simply by enabling the lazy attribute in the Department's hbm.xml mapping definition file, which will result in executing only the first statement from the *Employee* table and not the 5 queries from the *Department* table.

  ```
  <class name="com.Department"  table="Department" lazy="true" > …. </class>
  ```

  Only one query is required to return an employee object with its department initialized. In Hibernate, lazy loading of persistent objects are facilitated by proxies (i.e. virtual proxy design pattern). In the above example you have a *Department* object, which has a collection of *Employee* objects. Let's say that *Employee* objects are lazy loaded. If you make a call department.getEmployees() then Hibernate will load only the employeeIDs

and the version numbers of the *Employee* objects, thus saving loading of individual objects until later. So what you really have is a collection of proxies not the real objects. The reason being, if you have hundreds of employees for a particular department then chances are good that you will only deal with only a few of them. So, why unnecessarily instantiate all the *Employee* objects? This can be big performance in some situations.

- **Avoid N+1 Selects problem**: Having looked at the N+1 problem occurring inadvertently due to not having a lazy association in the previous example, now what if we need the Departmental information in addition to the Employee details. It is not a good idea to execute N+1 times.

```
<class name="com.Department"  table="Department" lazy="true" > …. </class>
```

Now to retrieve *Departmental* info you would:

```
Query query = session.createQuery("from Employee emp");
List list =  query.list();
Iterator it = list.iterator();

while(iter.hasNext()) {
    Employee emp = (Employee) it.next();
    emp.getDepartment().getName();       //N+1 problem. Since Department is not already loaded so
                                         //additional query is required for each department.
}
```

The solution is to make sure that the initial query retrieves all the data needed to load the objects by issuing a HQL fetch join (**eager loading**) as shown below:

```
"from Employee emp join fetch emp.Department dept"
```

The above HQL results in an inner join SQL as shown below:

```
SELECT <fields from Employee & Department> FROM employee
            inner join department on employee.departmentId = department.id.
```

Alternatively you can use a criteria query as follows:

```
Criteria crit = session.createCriteria(Employee.class);
Crit.setFetchMode("department", FetchMode.EAGER);
```

The above approach creates the following SQL:

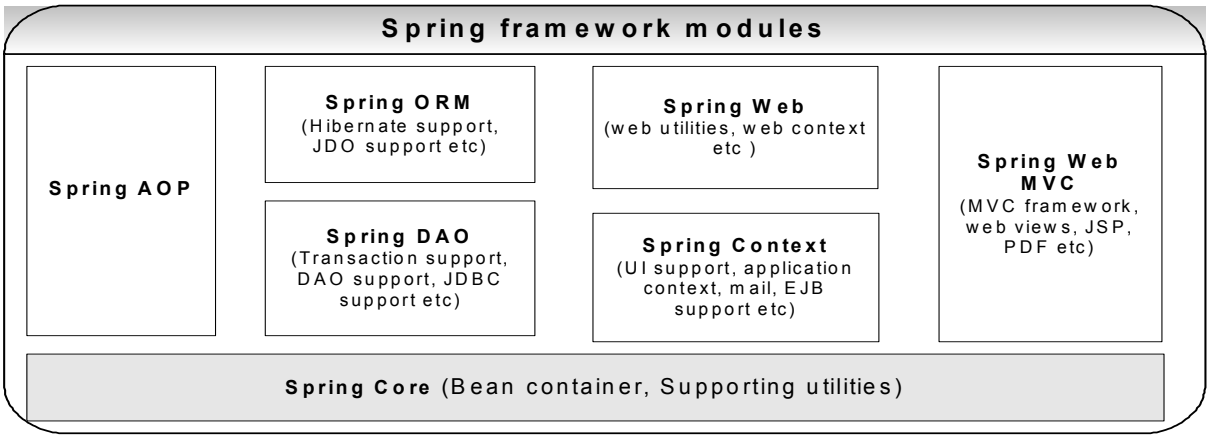
```
SELECT <fields from Employee & Department> FROM employee
            left outer join department on employee.departmentId = department.id where 1=1;
```

Q 17: Give an overview of the Spring framework?

A 17: The Spring framework is an open source and comprehensive framework for enterprise Java development. Unlike other frameworks, Spring does not impose itself on the design of a project due to its modular nature and, it has been divided logically into independent packages, which can function independently.

It includes abstraction layers for transactions, persistence frameworks, Web development, a JDBC integration framework, an AOP integration framework, email support, web services support etc. It also provides integration modules for popular Object-to-Relational (O/R) mapping tools like Hibernate, JDO etc. The designers of an application can feel free to use just a few Spring packages and leave out the rest. The other spring packages can be introduced into an existing application in a phased manner. Spring is based on the IOC pattern (aka Dependency Injection pattern) and also complements OOP (Object Oriented Programming) with AOP (Aspect Oriented Programming). You do not have to use AOP if you do not want to and AOP complements Spring IoC to provide a better middleware solution.

As shown in the diagram above the Spring modules are built on top of the core container, which defines how beans are configured, created and managed.



Core Container provides the essential basic functionality. The basic package in the spring framework is the org.springframework.beans package. The Spring framework uses JavaBeans and there are two ways in which clients can use the functionality of Spring framework -- BeanFactory and ApplicationContext. BeanFactory applies the **IOC pattern** and separates an application's configuration and dependency specification from the actual application code.

Spring Application Context is a configuration file that provides context information to Spring framework. The ApplicationContext builds on top of the BeanFactory and inherits all the basic features of Spring framework. In addition to basic features, ApplicationContext provides additional features like event management, internalization support, resource management, JNDI, EJB, email, and scheduling functionality. BeanFactory is useful in low memory situations, which does not have the excess baggage an ApplicationContext has. ApplicationContext assist the user to use Spring in a framework oriented way while the BeanFactory offers a programmatic approach.

Spring AOP enhances the Spring middleware support by providing declarative services. This allows any object managed by Spring framework to be AOP enabled. Spring AOP provides "declararative transaction management service" similar to transaction services provided by EJB. So with Spring AOP you can incorporate declarative transaction management without having to rely on EJB. AOP functionality is also fully integrated into Spring for logging and various other features.

Spring DAO uses org.springframework.jdbc package to provide all the JDBC related support required by your application.This abstraction layer offers a meaningful hierarchy for handling exceptions and errors thrown by different database vendors with the help of Spring's *SQLExceptionTranslator*. So this abstraction layer simplifies error handling and greatly reduces the amount of exception handling code you need to write. It also handles opening and closing of connections.

Spring ORM framework is built on top of Spring DAO and it plugs into several object-to-relational (ORM) mapping tools like Hibernate, JDO, etc. Spring provides very good support for Hibernate by supporting Hibernate sessions, Hibernate transaction management etc.

Spring Web sits on top of the ApplicationContext module to provide context for Web based applications. This provides integration with Struts MVC framework. Spring Web module also assists with binding HTTP request parameters to domain objects and eases the tasks of handling multipart requests.

Spring MVC framework provides a pluggable MVC architecture. The users have a choice to use this framework or continue to use other frameworks like Struts. Spring separates the roles of controller, model object, dispatcher and handler object, which makes it easier to customize. Spring Web framework does not force user to use only JSP, but accommodates various view technologies like XSLT, velocity templates, Tiles, etc.

Spring framework is a modular framework, which uses complementary technologies such as IoC and AOP to be used in complex enterprise application development. Spring functionality can be used in any J2EE server.

Q 18: How would EJB 3.0 simplify your Java development compared to EJB 1.x, 2.x?

A 18: EJB 3.0 is taking ease of development very seriously and has adjusted its model to offer the POJO (Plain Old Java Object) persistence and the new **O/R mapping model inspired by and based on Hibernate** (a less intrusive model). In EJB 3.0, **all kinds of enterprise beans are just POJOs**. EJB 3.0 **extensively uses Java annotations**, which replace excessive XML based configuration files and eliminate the need for rigid component

model used in EJB 1.x, 2.x. Annotations can be used to define a bean's business interface, O/R mapping information, resource references etc.

- In EJB 1.x, 2.x the container manages the behaviour and internal state of the bean instances at runtime. All the EJB 1.x, 2.x beans must adhere to a rigid specification. In EJB 3.0, all container services can be configured and delivered to any POJO in the application via annotations. **You can build complex object structures with POJOs. Java objects can inherit from each other.** EJB 3.0 components are only coupled via their published business interfaces hence the implementation classes can be changed without affecting rest of the application. This makes the application more robust, easier to test, more portable and makes it easier to build loosely coupled business components in POJO.

- EJB 3.0 unlike EJB 1.x, 2.x **does not have a home interface**. The bean class may or may not implement a business interface. If the bean class does not implement any business interface, a business interface will be generated using the public methods. If only certain methods should be exposed in the business interface, all of those methods can be marked with @BusinessMethod annotation.

- EJB 3.0 defines smart default values. For example by default all generated interfaces are local, but the @Remote annotation can be used to indicate that a remote interface should be generated.

- EJB 3.0 supports both unidirectional and bidirectional relationships between entities.

- EJB 3.0 makes use of dependency injection to make decoupled service objects and resources like queue factories, queues etc available to any POJO. Using the @EJB annotation, you can inject an EJB stub into any POJO managed by the EJB 3.0 container and using @Resource annotation you can inject any resource from the JNDI.

- EJB 3.0 wires runtime services such as transaction management, security, logging, profiling etc to applications at runtime. Since those services are not directly related to application's business logic they are not managed by the application itself. Instead, the services are transparently applied by the container utilizing AOP (Aspect Oriented Programming). To apply a transaction attribute to a POJO method using annotation:

```
public class Account {

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public getAccountDetails(){
        …
    }
}
```

- EJB QL queries can be defined through the @NamedQuery annotation. You can also create regular JDBC style queries using the *EntityManager*. POJOs are not persistent by birth and become persistent once it is associated with an *EntityManager.*

Q 19: Briefly explain key features of the JavaServer Faces (JSF) framework?

A 19: JavaServer Faces is a new framework for building Web applications using java. JSF provides you with the following main features:

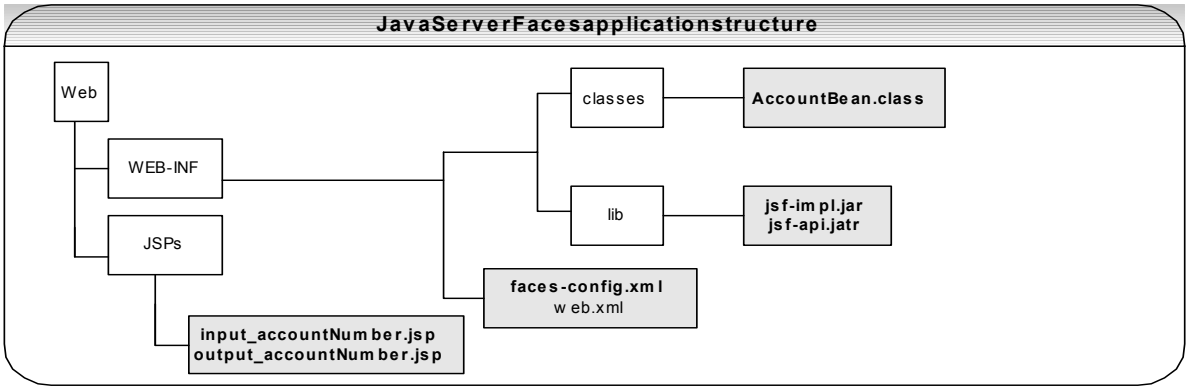
- Basic user interface components like buttons, input fields, links etc. and custom components like tree/table viewer, query builder etc. JSF was built with a component model in mind to allow tool developers to support Rapid Application Development (RAD). User interfaces can be created from these reusable server-side components.

- Provides a set of JSP tags to access interface components. Also provides a framework for implementing custom components.

- Supports mark up languages other than HTML like WML (Wireless Markup Language) etc by encapsulating event handling and component rendering. There is a single controller servlet every request goes through where the job of the controller servlet is to receive a faces page with components and then fire off events for each component render the components using a render tool kit.

- Uses a declarative navigation model by defining the navigation rules inside the XML configuration file faces-config.xml. This configuration file also defines bean resources used by JSF.

- JSF can hook into your model, which means the model is loosely coupled from JSF.



Let's look at some code snippets. Texts are stored in a properties file called **message.properties** so that this properties file can be quickly modified without having to modify the JSPs and also more maintainable because multiple JSP pages can use the same property.

```
account_nuber = Account number
account_button = Get account details
account_message=Processing account number :
```

input_accountNumber.jsp

```
<%@ taglib uri="http://java.sun.com.jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com.jsf/core" prefix="f" %>
<f:loadBundle basename="messages" var="msg"/>

<html>
  ...
  <body>
    <f:view>
      <h:form id="accountForm">
        <h:outputText value="#{msg.account_number}" />
        <h:inputText value="#{accountBean.accountNumber}" />
        <h:commandButton action="getAccount" value="#{msg.account_button}" />
      </h:form>
    </f:view>
  </body>
</html>
```

AccountBean.Java

```
public class AccountBean {
  String accountNumber;

  public String getAccountNumber() {
     return accountNumber;
  }

  public void setAccountNumber(String accountNumber) {
     this.accountNumber = accountNumber;
  }

}
```

faces-config.xml

```
...
<faces-config>

  <navigation-rule>
    <form-view-id>/jsps/input_accountNumber.jsp</form-view-id>
    <navigation-case>
      <from-outcome>getAccount</from-outcome>
      <to-view-id>/jsps/output_accountNumber.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
```

```
 ...

 <managed-bean>
  <managed-bean-name>accountBean</managed-bean-name>
  <managed-bean-class>AccountBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
 </managed-bean>

</faces-config>
```

output_accountNumber.jsp

```
<html>
 ...
 <body>
  <f:view>
   <h3>
    <h:outputText value="#{msg.account_message}" />
    <h:outputText value="#{accountBean.accountNumber}" />
   </h3>
  </f:view>
 </body>
</html>
```

Q 20: How would the JSF framework compare with the Struts framework?

A 20:

Struts framework	JavaServer Faces
Matured since Struts has been around for a few years. It has got several successful implementations.	JSF is in its early access release and as a result somewhat immature.
The heart of Struts framework is the Controller, which uses the Front Controller design pattern and the Command design pattern. Struts framework has got **only single event handler** for the HTTP request.	The heart of JSF framework is the Page Controller Pattern where there is a front controller servlet where all the faces request go through with the UI components and then fire off events for each component and render the components using a render toolkit. So JSF can have **several event handlers on a page**. Also JSF loosely couples your model, where it can hook into your model (i.e unlike Struts your model does not have to extend JSF classes).
Struts does not have the vision of Rapid Application Development (RAD).	JSF was built with a component model in mind to allow RAD. JSF can be thought of as a combination of Struts framework for thin clients and the Java Swing user interface framework for thick clients.
Has got flexible page navigation using navigation rules inside the struts-config.xml file and Action classes using maoping.findForward(…) .	JSF allows for more flexible navigation and a better design because the navigation rule (specified in **faces-config.xml**) is decoupled from the Action whereas Struts forces you to hook navigation into your Action classes.
Struts is a sturdy frame work which is extensible and flexible. The existing Struts applications can be migrated to use JSF component tags instead of the original Struts HTML tags because Struts tags are superseded and also not undergoing any active development. You can also use the **Struts-Faces Integration** library to migrate your pages one page at a time to using JSF component tags.	JSF is more flexible than Struts because it was able to learn from Struts and also extensible and can integrate with RAD tools etc. So JSF will be a good choice for new applications.

So far we have discussed some of the emerging paradigms (IOC (aka dependency injection), AOP, Annotations Oriented Programming, and O/R mapping) and some of the frameworks, which are based on these paradigms. These paradigms and frameworks simplify your programming model by hiding the complexities behind the framework and minimising the amount of code an application developer has to write. JSF is also gathering lot of momentum and popularity as a Web tier UI framework in new J2EE applications.

SECTION FIVE

Sample interview questions…

Tips:

- Try to find out the needs of the project in which you will be working and the needs of the people within the project.
- 80% of the interview questions are based on your own resume.
- Where possible briefly demonstrate how you applied your skills/knowledge in the key areas as described in this book. Find the right time to raise questions and answer those questions to show your strength.
- Be honest to answer technical questions, you are not expected to know everything (for example you might know a few design patterns but not all of them etc).
- Do not be critical, focus on what you can do. Also try to be humorous.
- Do not act in superior way.

Java

Questions	Hint
Multi-threading	
What language features are available to allow shared access to data in a multi-threading environment?	Synchronized block, Synchronized method, wait, notify
What is the difference between synchronized method and synchronized block? When would you use?	Block on subset of data. Smaller code segment.
What Java language features would you use to implement a producer (one thread) and a consumer (another thread) passing data via a stack?	wait, notify
Data Types	
What Java classes are provided for date manipulation?	Calendar, Date
What is the difference between String and StringBuffer?	mutable, efficient
How do you ensure a class is Serializable?	Implement Serializable
What is the difference between static and instance field of a class	Per class vs. Per Object
What method do you need to implement to store class in Hashtable or HashMap?	hashCode(), equals()
How do you exclude a field of the class from serialization?	transient
Inheritance	
What is the difference between an Interface and an abstract base class?	interface inheritance, implementation inheritance.
What does overriding a method mean? (What about overloading?)	inheritance (different signature)
Memory	
What is the Java heap, and what is the stack?	dynamic, program thread execution.
Why does garbage collection occur and when can it occur?	To recover memory, as heap gets full.
If I have a circular reference of objects, but I no longer reference any of them from any executing thread, will these cause garbage collection problems?	no
Exceptions	
What is the problem or benefits of catching or throwing type “java.lang.Exception”?	Hides all subsequent exceptions.
What is the difference between a runtime exception and a checked exception?	Must catch or throw checked exceptions.

Web components

Questions	HINT
JSP	
What is the best practice regarding the use of scriptlets in JSP pages? (Why?)	Avoid
How can you avoid scriptlet code?	custom tags, Java beans
What do you understand by the term JSP compilation?	compiles to servlet code
Servlets	
What does Servlet API provide to store user data between requests?	HttpSession
What is the difference between forwarding a request and redirecting?	redirect return to browser
What object do you use to forward a request?	RequestDispatcher
What do you need to be concerned about with storing data in a servlet instance fields?	Multi-threaded.
What’s the requirement on data stored in HttpSession in a clustered (distributable) environment?	Serializable
If I store an object in session, then change its state, is the state replicated to distributed Session?	No, only on setAttribute() call.
How does URL-pattern for servlet work in the web.xml?	/ddd/* or *.jsp
What is a filter, and how does it work?	Before/after request, chain.

Enterprise

Questions	Hint
JDBC	
What form of statement would you use to include user-supplied values?	PreparedStatement
Why might a preparedStatement be more efficient than a statement?	Execution plan cache.
How would you prevent an SQL injection attack in JDBC?	PreparsedStatement
What is the performance impact of testing against NULL in WHERE clause on Oracle?	Full table scan.
List advantages and disadvantages in using stored procedures?	Pro: integration with existing dbase, reduced network traffic Con: not portable, mutliple language knowledge required
What is the difference between sql.Date, sql.Time, and sql.Timestamp?	Date only, time only, date and time

If you had a missing int value how do you indicate this to PreparedStatement?	setNull(pos, TYPE)
How can I perform multiple inserts in one database interaction?	executeBatch
Given this problem: Program reads 100,000 rows, converts to Java class in list, then converts list to XML file using reflection. Runs out of program memory. How would you fix?	Read one row at time, limit select, allocate more heap (result set = cursor)
How might you model object inheritance in database tables?	Table per hierarchy, table per class, table per concrete class
JNDI	
What are typical uses for the JNDI API within an enterprise application	Resource management, LDAP access
Explain the difference between a lookup of these “java:comp/env/ejb/MyBean” and “ejb/MyBean”?	logical mapping performed for java:comp/env
What is difference between new InitialContext() from servlet or from an EJB?	Different JNDI environments initialised. ejb controller by ejb-jar.xml, servlet by web.xml
What is an LDAP server used for in an enterprise environment?	authentication, authorisation
What is authentication, and authorisation?	Confirming identity, confirming access rights
EJB	
What is the difference between Stateless and Stateful session beans (used?)	Stateless holds per client state
What is the difference between Session bean and Entity bean (when used?)	Entity used for persistence
With Stateless Session bean pooling, when would a container typically take a instance from the pool and when would it return it?	for each business method
What is difference between “Required”, “Supports”, “RequiresNew” “NotSupported”, “Mandatory”, “Never”	Needs transaction, existing OK but doesn’t need, must start new one, suspends transaction, must already be started, error if transaction
What is “pass-by-reference” and “pass-by-value”, and how does it affect J2EE applications?	Reference to actual object versus copy of object. RMI pass by value
What EJB patterns, best practices are you aware of? Describe at least two.	Façade, delegate, value list, DAO, value object
How do you define finder methods for a CMP?	Home, xml
If I reference an EJB from another EJB what can I cache to improve performance, and where should I do the caching?	Home, set it up in setSessionContext
Describe some issues/concerns you have with the J2EE specification	Get their general opinion of J2EE
Why is creating field value in setSessionContext of a performance benefit?	pooled, gc
What is difference between System exception and application exception from an EJB method?	System exception, container will auto rollback
What do you understand by the term “offline optimistic locking” or long-lived business transaction? How might you implement this using EJB?	version number, date, field comparisons
Explain performance difference between getting a list of summary information (e.g. customer list) via finder using a BMP entity vs Session using DAO?	BMP: n+1 database reads, n rmi calls
What is meant by a coarse-grained and a fine-grained interface?	Amount of data transferred per method call
XML/XSLT	
What is the difference between a DOM parser and a SAX parser?	DOM: reads entire model, SAX: event published during parsing
What is difference between DTD and XML Schema?	level of detail, Schema is in xml.
What does the JAXP API do for you?	Parser independence
What is XSLT and how can it be used?	xml translation
What would be the XPath to select any element called table with the class attribute of info?	Table[@class=’info’]
JMS	
How can asynchronous events be managed in J2EE?	JMS
How do transactions affect the onMessage() handling of a MDB?	Taking off queue
If you send a JMS message from an EJB, and transaction rollback, will message be sent?	yes
How do you indicate what topic or queue MDB should react to?	deployment descriptor
What is difference between a topic and a queue?	broadcast, single
SOAP	
What is a Web service, and how does it relate to SOAP?	SOAP is the protocol
What is a common transport for SOAP messages?	HTTP
What is WSDL? How would you use a WSDL file?	XML description of Web Service: interface and how to bind to it.
With new J2EE SOAP support what is: JAXR, JAX-RPC, and SAAJ?	registry, rpc, attachments
Security	
Where can container level security be applied in J2EE application?	Web URI’s, ejb methods
How can the current user be obtained in a J2EE application (Web and Enterprise)?	getUserPrincipal getCallerPrincipal
How can you perform role checks in a J2EE application (Web and enterprise)?	IsUserInRole()

	IsCallerInRole()

Design

Questions	Hint
OO	
Name some type of UML diagrams.	class, sequence, activity, use case
Describe some types of relationships can you show on class diagrams?	generalisation, aggregation, uses
What is difference between association, aggregation, and generalisation?	Relationship, ownership, inheritance
What is a sequence diagram used to display?	Object instance interactions via operations/signals
What design patterns do you use. Describe one you have used (not singleton)	e.g. Builder, Factory, Visitor, Chain of Command
Describe the observer pattern and an example of how it would be used	e.g. event notification when model changes to view
What are Use Cases?	Define interaction between actors and the system
What is your understanding of encapsulation?	Encapsulate data and behaviour within class
What is your understanding of polymorphism?	Class hierarchy, runtime determine instance
Process	
Have you heard of or used test-driven development?	e.g. XP process
What previous development process have you followed?	Rational, XP, waterfall
How do you approach capturing client requirements?	Numbered requirements, use case
What process steps would you include between the capture of requirements and when coding begins?	Architecture, Design, UML modelling
How would you go about solving performance issue in an application?	Set goals, establish bench, profile application, make changes one at a time
What developer based testing are you familiar with (before system testing?)	Unit test discussion
How might you test a business system exposed via a Web interface?	Automated script emulating browser
What is your experience with iterative development?	Multiple iteration before release
Distributed Application	
Explain a typical architecture of a business system exposed via Web interface?	Explain tiers (presentation, enterprise, resource) Java technology used in each tiers, hardware distribution of Web servers, application server, database server
Describe what tiers you might use in a typical large scale (> 200 concurrent users) application and the responsibilities of each tier (where would validation, presentation, business logic, persistence occur).	Another way of asking same question as above if their answer wasn't specific enough
Describe what you understand by being able to "scale" an application? How does a J2EE environment aid scaling.	Vertical and Horizontal scaling. Thread management, clustering, split tiers
What are some security issues in Internet based applications?	authentication, authorisation, data encryption, denial service, xss attacks

General

Questions	Hints
What configuration management are you familiar with?	e.g. CVS, ClearCase
What issue/tracking process have you followed?	Want details on bug recording and resolution process.
What are some key factors to working well within a team?	Gets a view on how you would work within interviewers' environment.
What attributes do you assess when considering a new job? (what makes it a good one)	Insight into what motivates you.
What was the last computing magazine you read? Last computing book? What is a regular online magazine/reference you use?	Understand how up to date you keep yourself.

GLOSSARY OF TERMS

TERM	DESCRIPTION
ACID	Atomicity, Consistency, Isolation, Duration.
aka	Also known as.
AOP	Aspect Oriented Programming
API	Application Program Interface
AWT	Abstract Window Toolkit
BLOB	Binary Large Object
BMP	Bean Managed Persistence
CGI	Common Gateway Interface
CLOB	Character Large OBject
CMP	Container Managed Persistence
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationships Management
CSS	Cascading Style Sheets
DAO	Data Access Object
DNS	Domain Name Service
DOM	Document Object Model
DTD	Document Type Definition
EAR	Enterprise ARchive
EIS	Enterprise Information System
EJB	Enterprise JavaBean
ERP	Enterprise Resource Planning
FDD	Feature Driven Development
GIF	Graphic Interchange Format
GOF	Gang Of Four
HQL	Hibernate Query Language.
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
IIOP	Internet Inter-ORB Protocol
IoC	Inversion of Control
IP	Internet Protocol
J2EE	Java 2 Enterprise Edition
JAAS	Java Authentication and Authorization Service
JAF	JavaBeans Activation Framework
JAR	Java ARchive
JAXB	Java API for XML Binding
JAXP	Java API for XML Parsing
JAXR	Java API for XML Registries
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML-based Web Services
JCA	J2EE Connector Architecture
JDBC	Java Database Connectivity
JDK	Java Development Kit
JMS	Java Messaging Service
JMX	Java Management eXtensions
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRMP	Java Remote Method Protocol
JSF	JavaServer Faces
JSP	Java Server Pages
JSTL	Java Standard Tag Library
JTA	Java Transaction API
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MOM	Message Oriented Middleware
MVC	Model View Controller
NDS	Novell Directory Service
NIO	New I/O
O/R mapping	Object to Relational mapping.
OO	Object Oriented

OOP	Object Oriented Programming
OOPL	Object Oriented Programming Language
ORB	Object Request Broker
ORM	Object to Relational Mapping.
POJI	Plain Old Java Interface
POJO	Plain Old Java Object
RAR	Resource adapter ARchive
RDBMS	Relational Database Management System
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RUP	Rational Unified Process
SAAJ	SOAP with attachment API for Java
SAX	Simple API for XML
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TDD	Test Driven Development
UDDI	Universal Description Discovery and Integration
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTF	
VO	Value Object which is a plain Java class which has attributes or fields and corresponding getter → getXXX() and setter → setXXX() methods .
WAR	Web ARchive
WSDL	Web Service Description Language
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XP	Extreme Programming
XPath	XML Path
XSD	XML Schema Definition
XSL	Extensible Style Language
XSL-FO	Extensible Style Language – Formatting Objects
XSLT	Extensible Style Language Transformation

RESOURCES

Articles

- Sun Java Certified Enterprise Architect by Leo Crawford on http://www.leocrawford.org.uk/work/jcea/part1/index.html.
- Practical UML: A Hands-On Introduction for Developers by Randy Miller on http://bdn.borland.com/article/0,1410,31863,00.html
- W3 Schools on http://www.w3schools.com/default.asp.
- LDAP basics on http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm?info/rzahy/rzahyovrco.htm.
- Java World articles on design patterns: http://www.javaworld.com/columns/jw-Java-design-patterns-index.shtml.
- Web Servers vs. App Servers: Choosing Between the Two By Nelson King on http://www.serverwatch.com/tutorials/article.php/1355131.
- Follow the Chain of Responsibility by David Geary on Java World - http://www.javaworld.com/javaworld/jw-08-2003/jw-0829-designpatterns.html.
- J2EE Design Patterns by Sue Spielman on http://www.onjava.com/pub/a/onjava/2002/01/16/patterns.html.
- The New Methodology by Martin Fowler on http://www.martinfowler.com/articles/newMethodology.html.
- Merlin brings nonblocking I/O to the Java platform by Aruna Kalagnanam and Balu G on http://www.ibm.com//developerworks/Java/library/j-javaio.
- Hibernate Tips and Pitfalls by Phil Zoio on http://www.realsolve.co.uk/site/tech/hib-tip-pitfall-series.php.
- Hibernate Reference Documentation on http://www.hibernate.org/hib_docs/reference/en/html_single/.
- Object-relation mapping without the container by Richard Hightower on http://www-128.ibm.com/developerworks/library/j-hibern/?ca=dnt515.
- Object to Relational Mapping and Relationships with Hibernate by Mark Eagle on http://www.meagle.com:8080/hibernate.jsp.
- Mapping Objects to Relational databases: O/R Mapping In detail by Scott W. Ambler on http://www.agiledata.org/essays/mappingObjects.html.
- I want my AOP by Ramnivas Laddad on Java World.
- Websphere Application Server 5.0 for iSeries – Performance Considerations by Jill Peterson.
- Dependency Injection using pico container by Subbu Ramanathan .
- Websphere Application Server & Database Performance tuning by Michael S. Pallos on http://www.bizforum.org/whitepapers/candle-5.htm.
- A beginners guide to Dependency Injection by Dhananjay Nene on http://www.theserverside.com/articles/article.tss?l=IOCBeginners.
- The Spring series: Introduction to the Spring framework by Naveen Balani on http://www-128.ibm.com/developerworks/web/library/wa-spring1.
- The Spring Framework by Benoy Jose.
- Inversion of Control Containersband the Dependency Injection pattern by Martin Fowler.
- Migrate J2EE Applications for EJB 3.0 by Debu Panda on JavaPro.
- EJB 3.0 in a nutshell by Anil Sharma on JavaWorld.
- Preparing for EJB 3.0 by Mike Keith on ORACLE Technology Network.
- Simplify enterprise Java development with EJB 3.0 by Michael Juntao Yuan on JavaWorld.
- J2SE: New I/O by John Zukowski on http://java.sun.com/developer/technicalArticles/releases/nio/.

- High-Performance I/O arrives by Danniel F. Savarese on JavaPro.
- Hibernate – Proxy Visitor Pattern by Kurtis Williams.
- Best Practices for Exception Handling by Gunjan Doshi.
- Three Rules for Effective Exception Handling by Jim Cushing.
- LDAP and JNDI: Together forever – by Sameer Tyagi.
- Introduction To LDAP – by Brad Marshall.
- Java theory and practice: Decorating with dynamic proxies by Brian Goetz.
- Java Dynamic Proxies: One Step from Aspect-Oriented Programming by Lara D'Abreo.
- Java Design Patterns on http://www.allapplabs.com/java_design_patterns .
- Software Design Patterns on http://www.dofactory.com/Patterns/Patterns.aspx .
- JRun: Core Dump and Dr. Watson Errors on http://www.macromedia.com/cfusion/knowledgebase/index.cfm?id=tn_17534
- www.javaworld.com articles.
- http://www-128.ibm.com/developerworks/java articles.
- http://www.devx.com/java articles.
- www.theserverside.com/tss articles.
- http://javaboutique.internet.com/articles articles.

Books

- Beginning Java 2 by Ivor Horton.
- Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (GoF) .
- UML Distilled by Martin Fowler, Kendall Scott .
- Mastering Enterprise Java Beans II by Ed Roman, Scott Ambler, Tyler Jewell, Floyd Marinescu [download for free] .
- EJB Design Patterns by Floyd Marinescu [download for free] .
- Sun Certified Enterprise Architect for J2EE Technology Study Guide by Mark Cade and Simon Roberts.
- Professional Java Server Programming - J2EE edition by Wrox publication.
- Design Patterns Java Companion by James W. Cooper (Free download: http://www.patterndepot.com/put/8/JavaPatterns.htm).
- Test Driven Development – By Example, by Kent Beck.

INDEX

Emerging Technologies/Frameworks

Briefly explain key features of the JavaServer Faces (JSF) framework? 223
Explain Object-to-Relational (O/R) mapping? 218
Explain some of the pitfalls of Hibernate and explain how to avoid them? 220
Give an overview of hibernate framework? 218
Give an overview of the Spring framework? 221
How would EJB 3.0 simplify your Java development compared to EJB 1.x, 2.x? 222
How would the JSF framework compare with the Struts framework? 225
What are the benefits of IOC (aka Dependency Injection)? 217
What are the differences between OOP and AOP? 214
What are the different types of dependency injections? 217
What are the pros and cons of annotations over XML based deployment descriptors? 215
What is aspect oriented programming? Explain AOP? 212
What is attribute or annotation oriented programming? 215
What is inversion of control (IOC) (also known as dependency injection)? 216
What is Test Driven Development (TDD)? 211
What is the difference between a service locator pattern and an inversion of control pattern? 217
What is the point of Test Driven Development (TDD)? 211
What is XDoclet? 216
Why dependency injection is more elegant than a JNDI lookup to decouple client and the service? 218

Enterprise - Best practices and performance considerations

Explain some of the J2EE best practices to improve performance? 141
Explain some of the J2EE best practices? 139
Give some tips on J2EE application server performance tuning? 139

Enterprise - EJB 2.x

Can an EJB client invoke a method on a bean directly? 99
Discuss EJB container security? 105
Explain EJB architecture? 96
Explain exception handling in EJB? 103
Explain lazy loading and dirty marker strategies? 109
How can we determine if the data is stale (for example when using optimistic locking)? 104
How do you rollback a container managed transaction in EJB? 103
How to design transactional conversations with session beans? 102
What are EJB best practices? 106
What are isolation levels? 101
What are not allowed within the EJB container? 105
What are the implicit services provide by an EJB container? 101
What are transactional attributes? 101
What is a business delegate? Why should you use a business delegate? 107
What is a distributed transaction? What is a 2-phase commit? 102
What is a fast-lane reader? 109
What is a Service Locator? 109
What is a session façade? 108
What is a value object pattern? 108
What is dooming a transaction? 102
What is the difference between Container Managed Persistence (CMP) and Bean Managed Persistence (BMP)? 99
What is the difference between EJB 1.1 and EJB 2.0? What is the difference between EJB 2.x and EJB 3.0? 100
What is the difference between EJB and JavaBeans? 95
What is the difference between optimistic and pessimistic concurrency control? 104
What is the difference between session and entity beans? 99
What is the difference between stateful and stateless session beans? 99
What is the role of EJB 2.x in J2EE? 95

Enterprise - J2EE

Explain J2EE class loaders? 68
Explain MVC architecture relating to J2EE? 63
Explain the J2EE 3-tier or n-tier architecture? 61
So what is the difference between a component and a service you may ask? 60
What are ear, war and jar files? What are J2EE Deployment Descriptors? 64
What is J2EE? What are J2EE components and services? 60
What is the difference between a Web server and an application server? 64
Why use design patterns in a J2EE application? 64

Enterprise - JDBC

Explain differences among java.util.Date, java.sql.Date, java.sql.Time, and java.sql.Timestamp? 86
How to avoid the “running out of cursors” problem? 85
What are JDBC Statements? What are different types of statements? How can you create them? 83
What is a Transaction? What does setAutoCommit do? 84
What is JDBC? How do you connect to a database? 83
What is the difference between JDBC-1.0 and JDBC-2.0? What are Scrollable ResultSets, Updateable ResultSets, RowSets, and Batch updates? 85
What is the difference between statements and prepared statements? 86

Enterprise - JMS

Discuss some of the design decisions you need to make regarding your message delivery? 112
Give an example of a J2EE application using Message Driven Bean with JMS? 114
How JMS is different from RPC? 110
What are some of the key message characteristics defined in a message header? 111
What is Message Oriented Middleware? What is JMS? 110
What type of messaging is provided by JMS? 111

Enterprise - JNDI & LDAP

Explain the difference between the look up of “java comp/env/ejb/MyBean” and “ejb/MyBean”? 88
Explain the RMI architecture? 90
How will you pass parameters in RMI? 93
What are the differences between RMI and a socket? 92
What are the services provided by the RMI Object? 92
What is a JNDI InitialContext? 88
What is a remote object? Why should we extend UnicastRemoteObject? 91
What is an LDAP server? And what is it used for in an enterprise environment? 88
What is HTTP tunnelling or how do you make RMI calls across firewalls? 93

What is JNDI? And what are the typical uses within a J2EE application? 87
What is the difference between RMI and CORBA? 92
Why use LDAP when you can do the same with relational database (RDBMS)? 89

Enterprise - JSP

Explain hidden and output comments? 80
Explain the life cycle methods of a JSP? 78
How will you avoid scriptlet code in JSP? 83
Is JSP variable declaration thread safe? 80
Tell me about JSP best practices? 82
What are custom tags? Explain how to build custom tags? 81
What are implicit objects and list them? 79
What are the differences between static and a dynamic include? 79
What are the different scope values or what are the different scope values for <jsp usebean> ? 79
What are the main elements of JSP? What are scriplets? What are expressions? 78
What is a JSP? What is it used for? What do you understand by the term JSP translation phase or compilation phase? 77
What is a TagExtraInfo class? 82
What is the difference between custom JSP tags and Javabeans? 82

Enterprise - Logging, testing and deployment

Enterprise - Logging, testing and deployment 143
Give an overview of log4J? 141
How do you initialize and use Log4J? 142
What is the hidden cost of parameter construction when using Log4J? 142
What is the test phases and cycles? 143

Enterprise - Personal

Have you used any load testing tools? 144
Tell me about yourself or about some of the recent projects you have worked with? What do you consider your most significant achievement? Why do you think you are qualified for this position? Why should we hire you and what kind of contributions will you make? 144
What operating systems are you comfortable with? 144
What source control systems have you used? 144
Which on-line technical resources do you use to resolve any design and/or development issues? 144

Enterprise - RUP & UML

Explain the 4 phases of RUP? 127
What are the characteristics of RUP? Where can you use RUP? 128
What are the different types of UML diagrams? 128
What is RUP? 126
What is the difference between a collaboration diagram and a sequence diagram? 133
What is the difference between aggregation and composition? 133
When to use 'use case' diagrams? 128
When to use activity diagrams? 132
When to use class diagrams? 129
When to use interaction diagrams? 131
When to use object diagrams? 130
When to use package diagrams? 130
When to use statechart diagram? 131
Why is UML important? 128

Enterprise - Servlet

Briefly discuss the following patterns Composite view, View helper, Dispatcher view and Service to worker? Or explain J2EE design patterns? 76
Explain declarative security for WEB applications? 74
Explain Servlet URL mapping? 77
Explain the directory structure of a WEB application? 71
Explain the Front Controller design pattern or explain J2EE design patterns? 75
Explain the life cycle methods of a servlet? 70
How do you make a Servlet thread safe? What do you need to be concerned about with storing data in Servlet instance fields? 72
HTTP is a stateless protocol, so how do you maintain state? How do you store user data between requests? 69
If an object is stored in a session and subsequently you change the state of the object, will this state change replicated to all the other distributed sessions in the cluster? 74
What are the considerations for servlet clustering? 73
What are the ServletContext and ServletConfig objects? What are Servlet environment objects? 72
What is a filter, and how does it work? 74
What is a RequestDispatcher? What object do you use to forward a request? 73
What is pre-initialization of a Servlet? 73
What is the difference between CGI and Servlet? 69
What is the difference between doGet () and doPost () or GET and POST? 71
What is the difference between forwarding a request and redirecting a request? 73
What is the difference between HttpServlet and GenericServlet? 72

Enterprise - Software development process

What software development processes/principles are you familiar with? 144

Enterprise - SQL, Tuning and O/R mapping

Explain a sub-query? How does a sub-query impact on performance? 121
Explain inner and outer joins? 119
How can you performance tune your database? 122
How do you implement one-to-one, one-to-many and many-to-many relationships while designing tables? 122
How do you map inheritance class structure to relational data model? 123
How will you map objects to a relational database? How will you map class inheritance to relational data model? 122
What is a view? Why will you use a view? What is an aggregate function? 124
What is normalization? When to denormalize? 121

Enterprise - Struts

Are Struts action classes thread-safe? 135
Give an overview of Struts? 133
How do you implement internationalization in Struts? 136
How do you upload a file in Struts? 135
What design patterns are used in Struts? 136
What is a synchronizer token pattern in Struts or how will you protect your Web against multiple submissions? 135
What is an action mapping in Struts? How will you extend Struts? 136

Enterprise - Web and Application servers

Explain Java Management Extensions (JMX)? 138
What application servers, Web servers, LDAP servers, and Database servers have you used? 137
What is a virtual host? 137
What is application server clustering? 138
What is the difference between a Web server and an application server? 137

Enterprise - Web and Applications servers

Explain some of the portability issues between different application servers? 139

Enterprise - XML

What is the difference between a SAX parser and a DOM parser? 115
What is XML? And why is XML important? 114
What is XPATH? What is XSLT/XSL/XSL-FO/XSD/DTD etc? What is JAXB? What is JAXP? 115
Which is better to store data as elements or as attributes? 115

How would you go about...?
How would you go about applying the design patterns in your Java/J2EE application? 165
How would you go about applying the Object Oriented (OO) design concepts in your Java/J2EE application? 160
How would you go about applying the UML diagrams in your Java/J2EE project? 162
How would you go about describing the open source projects like JUnit (unit testing), Ant (build tool), CVS (version control system) and log4J (logging tool) which are integral part of most Java/J2EE projects? 199
How would you go about describing the software development processes you are familiar with? 163
How would you go about describing Web services? 206
How would you go about designing a Java/J2EE application? 153
How would you go about determining the enterprise security requirements for yor Java/J2EE application? 194
How would you go about documenting your Java/J2EE application? 152
How would you go about identifying any potential thread-safety issues in your Java/J2EE application? 158
How would you go about identifying any potential transactional issues in your Java/J2EE application? 159
How would you go about identifying performance and/or memory issues in your Java/J2EE application? 156
How would you go about improving performance in your Java/J2EE application? 157
How would you go about minimising memory leaks in your Java/J2EE application? 157
Java
Briefly explain high-level thread states? 38
Discuss the Java error handling mechanism? What is the difference between Runtime (unchecked) exceptions and checked exceptions? What is the implication of catching all the exceptions with the type “Exception”? 35
Explain different ways of creating a thread? 38
Explain Java class loaders? Explain dynamic class loading? 13
Explain Outer and Inner classes (or Nested classes) in Java? When will you use an Inner Class? 31
Explain static vs dynamic class loading? 13
Explain the assertion construct? 19
Explain the Java Collection framework? 21
Explain the Java I/O streaming concept and the use of the decorator design pattern in Java I/O? 26
Explain threads blocking on I/O? 41
Give a few reasons for using Java? 12
Give an example where you might use a static method? 29
How can threads communicate with each other? How would you implement a producer (one thread) and a consumer (another thread) passing data (via stack)? 40
How can you improve Java I/O performance? 28
How do you express an ‘is a’ relationship and a ‘has a’ relationship or explain inheritance and composition? What is the difference between composition and aggregation? 15
How does Java allocate stack and heap memory? Explain re-entrant, recursive and idempotent methods/functions? 31
How does the Object Oriented approach improve software development? 14
How does thread synchronization occurs inside a monitor? What levels of synchronization can you apply? What is the difference between synchronized method and synchronized block? 39
How will you call a Web server from a stand alone Java application? 44
If 2 different threads hit 2 different synchronized methods in an object at the same time will they both continue? 40
If you have a circular reference of objects, but you no longer reference it from an execution thread, will this object be a potential candidate for garbage collection? 34
What are “static initializers” or “static blocks with no function names”? 14
What are access modifiers? 30
What are some of the best practices relating to Java collection? 22
What are the advantages of Object Oriented Programming Languages (OOPL)? 14
What are the benefits of the Java collection framework? 22
What do you know about the Java garbage collector? When does the garbage collection occur? Explain different types of references in Java? 34
What do you mean by polymorphism, inheritance, encapsulation, and dynamic binding? 15
What is a daemon thread? 40
What is a factory pattern? 42
What is a final modifier? Explain other Java modifiers? 30
What is a singleton pattern? How do you code it in Java? 41
What is a socket? How do you facilitate inter process communication in Java? 43
What is a user defined exception? 37
What is design by contract? Explain the assertion construct? 18
What is serialization? How would you exclude a field of a class from serialization or what is a transient variable? What is the common use? 26
What is the difference between aggregation and composition? 15
What is the difference between an abstract class and an interface and when should you use them? 20
What is the difference between an instance variable and a static variable? Give an example where you might use a static variable? 29
What is the difference between C++ and Java? 12
What is the difference between final, finally and finalize() in Java? 31
What is the difference between processes and threads? 37
What is the difference between yield and sleeping? 39
What is the main difference between a String and a StringBuffer class? 25
What is the main difference between an ArrayList and a Vector? What is the main difference between Hashmap and Hashtable? 21
What is the main difference between pass-by-reference and pass-by-value? 25
What is the main difference between shallow cloning and deep cloning of objects? 29
What is the main difference between the Java platform and the other software platforms? 12
What is type casting? Explain up casting vs down casting? When do you get ClassCastException? 33
When is a method said to be overloaded and when is a method said to be overridden? 21
When providing a user defined key class for storing objects in the Hashmaps or Hashtables, what methods do you have to provide or override (ie method overriding)? 24
When to use an abstract class? 20
When to use an interface? 21
Where and how can you use a private constructor? 30
Why is it not advisable to catch type “Exception”? 36
Why should you catch a checked exception late in a catch {} block? 36
Why should you throw an exception early? 36
Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java? 21
Java - Applet
How will you communicate between two Applets? 49
How will you initialize an applet? 48
How would you communicate between applets and servlets? 49
What is a signed Applet? 49

What is the difference between an applet and an application? 49
What is the order of method invocation in an applet? 48
Java - Performance & Memory leaks
How would you detect and minimise memory leaks in Java? 51
How would you improve performance of a Java application? 50
Why does the JVM crash with a core dump or a Dr.Watson error? 52
Java - Swing
Explain layout managers? 47
Explain the Swing Action architecture? 45
Explain the Swing delegation event model? 48
Explain the Swing event dispatcher mechanism? 46
If you add a component to the CENTER of a border layout, which directions will the component stretch? 45
What do you understand by MVC as used in a JTable? 47
What is the base class for all Swing components? 45
What is the difference between AWT and Swing? 44
Java/J2EE - Personal
Did you have to use any design patterns in your Java project? 53
Do you have any role models in software development? 55
How do you handle pressure? Do you like or dislike these situations? 54
Tell me about yourself or about some of the recent projects you have worked with? What do you consider your most significant achievement? Why do you think you are qualified for this position? Why should we hire you and what kind of contributions will you make? 53
What are your career goals? Where do you see yourself in 5-10 years? 55
What are your strengths and weaknesses? Can you describe a situation where you took initiative? Can you describe a situation where you applied your problem solving skills? 54
What do you like and/or dislike most about your current and/or last position? 54
What past accomplishments gave you satisfaction? What makes you want to work hard? 55
What was the last Java related book or article you read? 55
Why are you leaving your current position? 54
Why do you want to work for us? 55
Key Points
Enterprise - Key Points 146
Java - Key Points 56