Learn Java/J2EE core concepts and key areas

With

Java/J2EE Job Interview Companion

By

K.Arulkumaran
&
A.Sivayini

Technical Reviewers

Craig Malone
Stuart Watson
Arulazi Dhesiaseelan
Lara D'Albreo

Cover Design, Layout, & Editing

A.Sivayini

Acknowledgements

A. Sivayini
Mr. & Mrs. R. Kumaraswamipillai

Java/J2EE
Job Interview Companion

 ISBN 978-1-4116-6824-9

Please e-mail feedback & corrections (technical, grammatical and/or spelling) to
java-interview@hotmail.com

To purchase the complete book: http://www.lulu.com/content/192463
(2nd edition)

Home page: http://www.lulu.com/java-success

First Edition (220+ Q&A): **Dec 2005**
Second Edition (400+ Q&A): **March 2007**

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** ▪ Fundamentals ▪ Swing ▪ Applet ▪ Performance and Memory issues ▪ Personal and Behavioral/Situational ▪ Behaving right in an interview ▪ Key Points
SECTION 2	Interview questions and answers on: **Enterprise Java** ▪ J2EE Overview ▪ Servlet ▪ JSP ▪ JDBC / JTA ▪ JNDI / LDAP ▪ RMI ▪ EJB ▪ JMS ▪ XML ▪ SQL, Database, and O/R mapping ▪ RUP & UML ▪ Struts ▪ Web and Application servers. ▪ Best practices and performance considerations. ▪ Testing and deployment. ▪ Personal and Behavioral/Situational ▪ Key Points
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 minimizing 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 designing a Web application where the business tier is on a separate machine from the presentation tier. The business tier should talk to 2 different databases and your design should point out the different design patterns? 13. How would you go about determining the enterprise security requirements for your Java/J2EE application? 14. 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? 15. How would you go about describing Service Oriented Architecture (**SOA**) and 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

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), Dependency Injection DI (aka IoC – Inversion of Control) 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 key areas, 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 categorized 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 key areas 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 key areas**" or "**Java/J2EE Job Interview Companion**". The reason I chose "**Java/J2EE Job Interview Companion**" is because the core concepts and key areas discussed in this book helped me to be successful in my interviews, helped me to survive and succeed at my work regardless what my job (junior developer, senior developer, technical lead, designer, contractor etc) was and also gave me thumbs up in code reviews. This book also has been set out as a handy reference guide and a roadmap for building enterprise Java applications.

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 organizations 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 organization, 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 **modeling 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 **Q7**, **Q8**, **Q9**, and **Q10** 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 **Q72 – Q74** in Java section and **Q123, Q125** in Enterprise section.

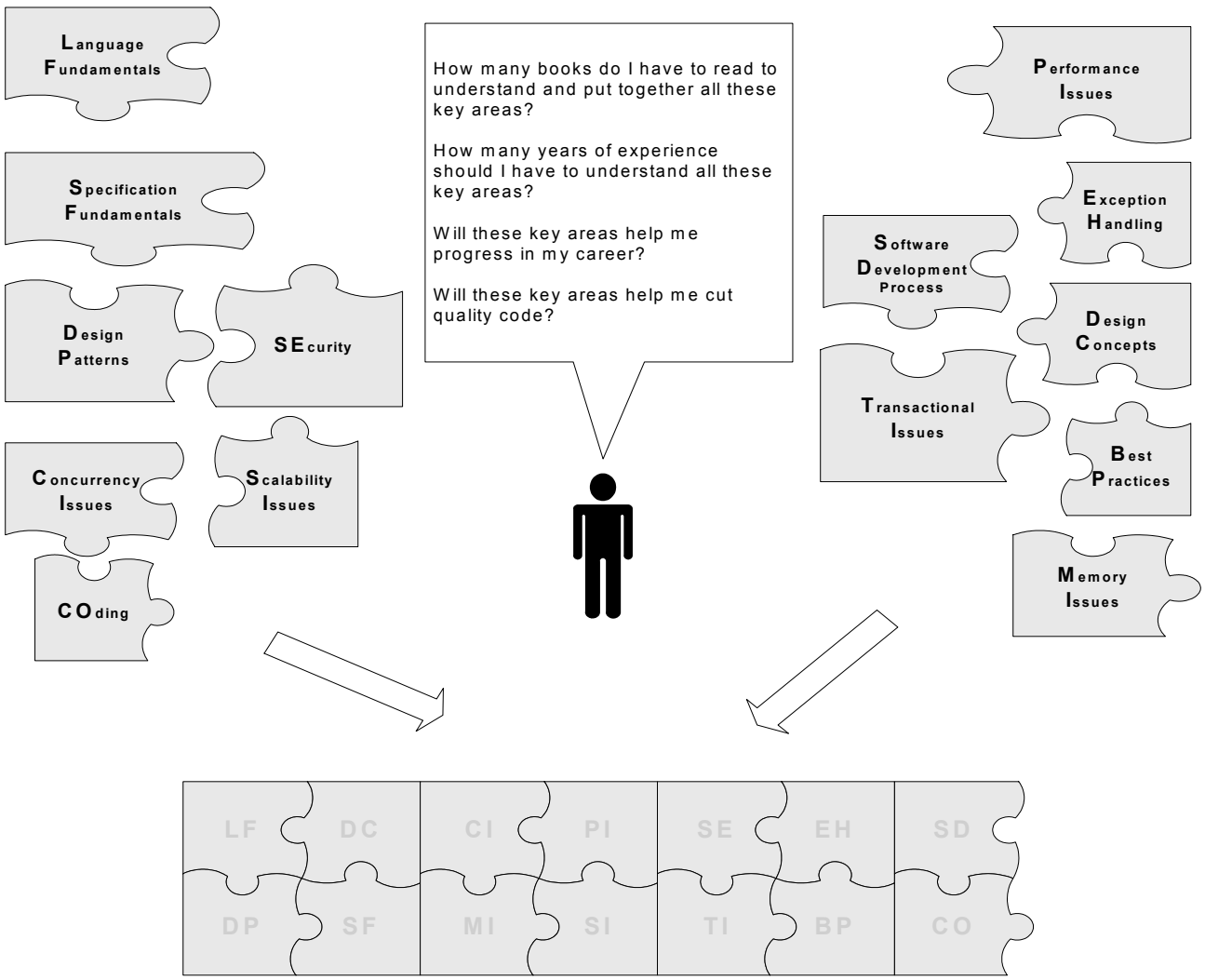
If you happen to be in an interview with an organization which is working on a pilot project using a different development methodology like agile methodology etc or has just started adopting a newer **development process** or **methodology** then you are likely to be asked questions on this key area.

If the team lead/architect of the organization you are being interviewed for feels that the current team is lacking skills in the key areas of **design concepts** and **design patterns** then you are likely to be asked questions on these key areas.

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 favorite 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 **Q10** in Java section.
- Program to interfaces not to implementations: Refer **Q12, Q17** in Java section.
- Use of relevant design patterns: Refer **Q11, Q12** in How would you go about… section.
- Use of Java collections API and exceptions correctly: Refer **Q16** and **Q39** in Java section.
- Stay away from hard coding values: Refer **Q05** in Java section.

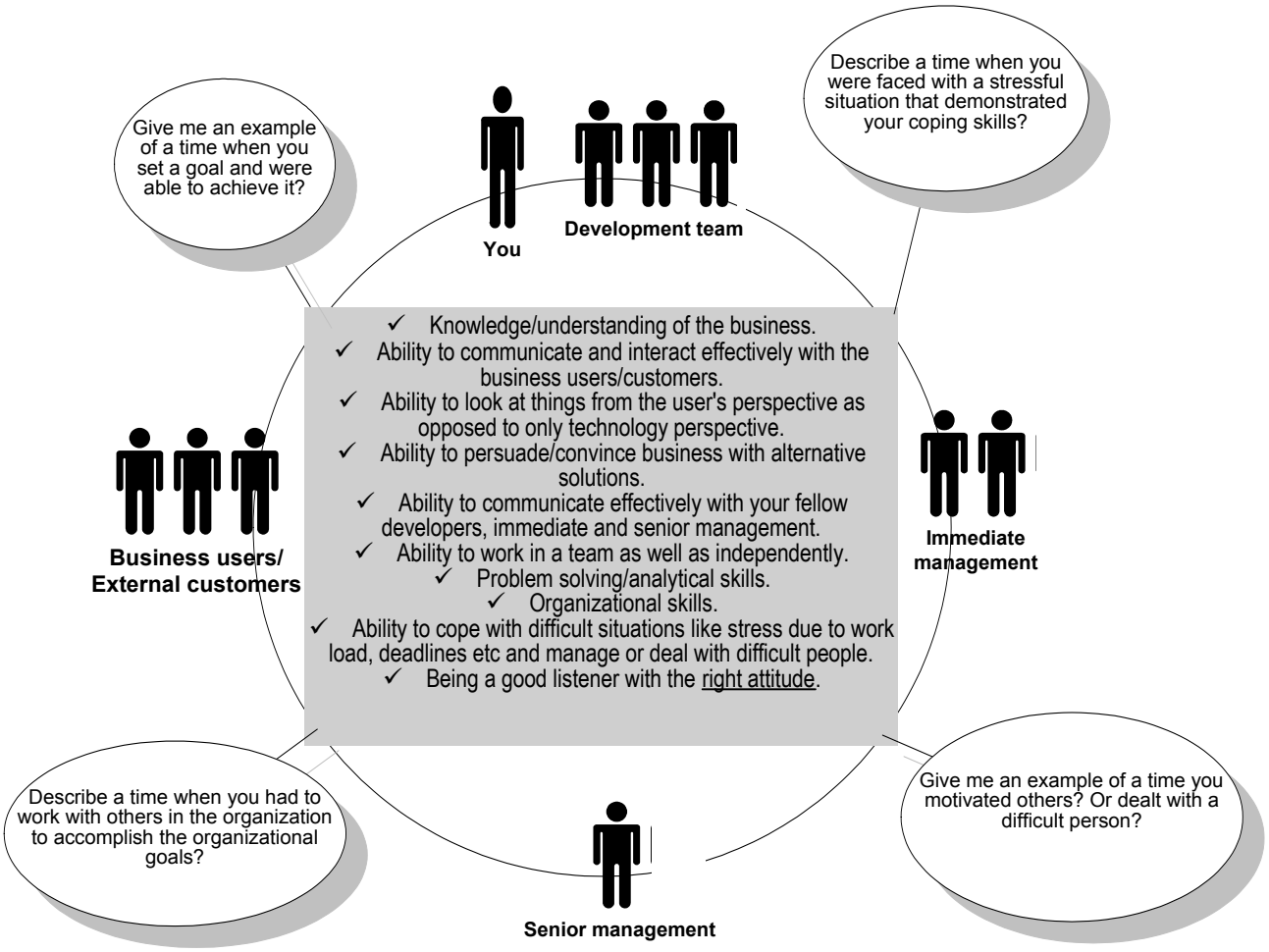


This book aims to solve the above dilemma.

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 organizations is to understand and proactively apply and

resolve the issues relating to the **Key Areas** discussed in this book. 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 humor.**

"Technical skills must be complemented with good business and interpersonal skills."



IMPORTANT: Technical skills alone are not sufficient for you to perform well in your interviews and progress in your career. Your technical skills **must be complemented** with business skills (i.e. knowledge/understanding of the business, ability to communicate and interact effectively with the business users/customers, ability to look at things from the users' perspective as opposed to only from technology perspective, ability to persuade/convince business with alternative solutions, which can provide a win/win solution from users' perspective as well as technology perspective), ability to communicate effectively with your fellow developers, immediate and senior management, ability to work in a team as well as independently, problem solving/analytical skills, organizational skills, ability to cope with difficult situations like stress due to work load, deadlines etc and manage or deal with difficult people, being a good listener with the right attitude (It is sometimes possible to have "**I know it all attitude**", when you have strong technical skills. These are discussed in "**Java – Personal"** and "**Enterprise Java – Personal"** sub-sections with examples.

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

-- Read/Browse all questions marked as **"FAQ"** in all four sections.
-- Read/Browse **Key Points** in Java and Enterprise Java sections.

Key Areas Index

I have categorized 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 Java section**	**How would you go about…?**	**Emerging Technologies / Frameworks**
Language Fundamentals	LF	Q1-Q6, Q12-Q16, Q18-Q24, Q26-Q33, Q35-Q38, Q41-Q50, Q53-Q71	-		Q10, Q15, Q17, Q19
Specification Fundamentals	SF	-	Q1, Q2, Q4, Q6, Q7-Q15, Q17-Q19, Q22, Q26-Q33, Q35-Q38, Q41, Q42, Q44, Q46-Q81, Q89-Q93, Q95-Q97, Q99, 102, Q110, Q112-Q115, Q118-Q119, Q121, Q126, Q127, Q128	Q15	
Design Concepts	DC	Q1, Q7-Q12, Q15, Q26, Q22, Q56	Q2, Q3, Q19, Q20, Q21, Q31, Q45, Q91, Q94, Q98, Q101, Q106, Q107, Q108, Q109, Q111	Q02, Q08, Q09, Q15	Q3 - Q13, Q13, Q14, Q16, Q17, Q18, Q20
Design Patterns	DP	Q12, Q16, Q24, Q36, Q51, Q52, Q58, Q63, Q75	Q5, Q5, Q22, Q24, Q25, Q41, Q83, Q84, Q85, Q86, Q87, Q88, Q110, Q111, Q116	Q11, Q12	Q9 - Q13
Transactional Issues	TI	-	Q43, Q71, Q72, Q73, Q74, Q75, Q77, Q78, Q79	Q7	
Concurrency Issues	CI	Q15, Q17, Q21, Q34, Q42, Q46, Q62	Q16, Q34, Q72, Q78, Q113	Q6	
Performance Issues	PI	Q15, Q17,Q20-Q26, Q46, Q62, Q72	Q10, Q16, Q43, Q45, Q46, Q72, Q83-Q88, Q93, Q97, Q98, Q100, Q102, Q123, Q125, Q128	Q3, Q5	
Memory Issues	MI	Q26, Q34, Q37,Q38, Q42, Q51, Q73, Q74	Q45, Q93	Q3, Q4	
Scalability Issues	SI	Q23, Q24	Q20, Q21, Q120, Q122		
Exception Handling	EH	Q39, Q40	Q76, Q77		
Security	SE	Q10, Q35, Q70	Q12, Q13, Q23, Q35, Q46, Q51, Q58, Q81, Q92	Q13	
Best Practices	BP	Q17, Q25, Q39, Q72, Q73	Q10, Q16, Q39, Q40, Q41, Q46, Q82, Q124, Q125	Q1, Q2	

Software Development Process	SD	-	Q103-Q109, Q129, Q130, Q132, Q136	Q1, Q9, Q10, Q14	Q1, Q2
Coding[1]	CO	Q05, Q10, Q12, Q14 – Q21, Q23, Q25, Q26, Q33, Q35, Q39, Q51, Q52, Q55	Q10, Q18, Q21, Q23, Q36, Q38, Q42, Q43, Q45, Q74, Q75, Q76, Q77, Q112, Q114, Q127, Q128	Q11, Q12	
Frequently Asked Questions	FAQ	Q1, Q6, Q7, Q9, Q10, Q12, Q13, Q14, Q15, Q16, Q18, Q20, Q21, Q22, Q23, Q27, Q28, Q29, Q30, Q31, Q32, Q36, Q37, Q43, Q45, Q46, Q48, Q51, Q52, Q55, Q58, Q60, Q62, Q63, Q64, Q67, Q68, Q69, Q70, Q71 Q72 – Q86	Q1, Q2, Q3, Q7, Q10, Q11, Q12, Q13, Q16, Q19, Q22, Q24, Q25, Q27, Q28, Q30, Q31, Q32, Q34, Q35, Q36, Q39, Q40, Q41, Q42, Q43, Q45, Q46, Q48, Q49, Q50, Q52, Q53, Q61, Q63, Q65, Q66, Q69, Q70, Q71, Q72, Q73, Q76, Q77, Q82, Q83, Q84, Q85, Q86, Q87, Q90, Q91, Q93, Q95, Q96, Q97, Q98, Q100, Q101, Q102, Q107, Q108, Q110, Q113, Q115, Q116, Q118, Q123, Q124, Q125, Q126, Q129, Q130, Q131, Q133, Q134, Q135, Q136.	Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q12, Q15	Q1, Q6, Q7, Q9, Q10, Q11, Q15, Q16, Q17, Q18

[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

FAQ - Frequently **A**sked **Q**uestions

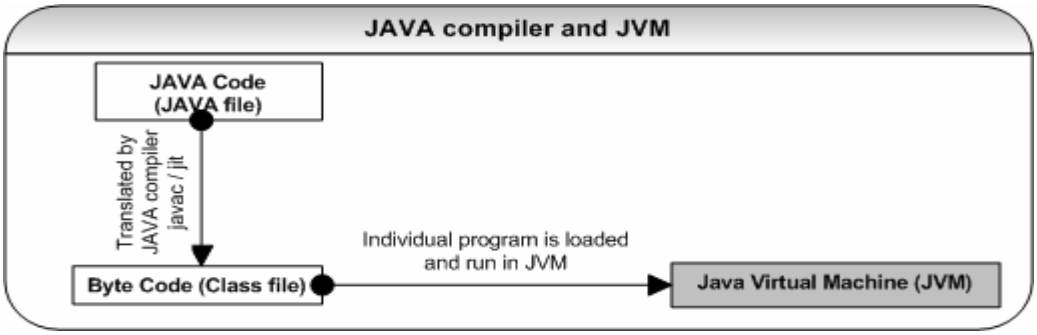
[1] Unlike other key areas, the CO is not always shown against the question but shown above the actual content of relevance within a question.

Java – Fundamentals

Q 01: Give a few reasons for using Java? LF DC FAQ
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 standardized APIs (Application Programming 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**) – set of classes written using the Java language and run on the JVM.

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 **Q9** and **Q10** in Java section.]
- Java does not support destructors but 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 Collections Framework – Refer **Q16**, **Q17** 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 **Q37** in Java section].

Q 04: What are the usages of Java packages? **LF**

A 04: It helps resolve naming conflicts when different packages have classes with the same names. This also helps you organize files within your project. **For example:** **java.io** package do something related to I/O and **java.net** package do something to do with network and so on. If we tend to put all .java files into a single package, as the project gets bigger, then it would become a nightmare to manage all your files.

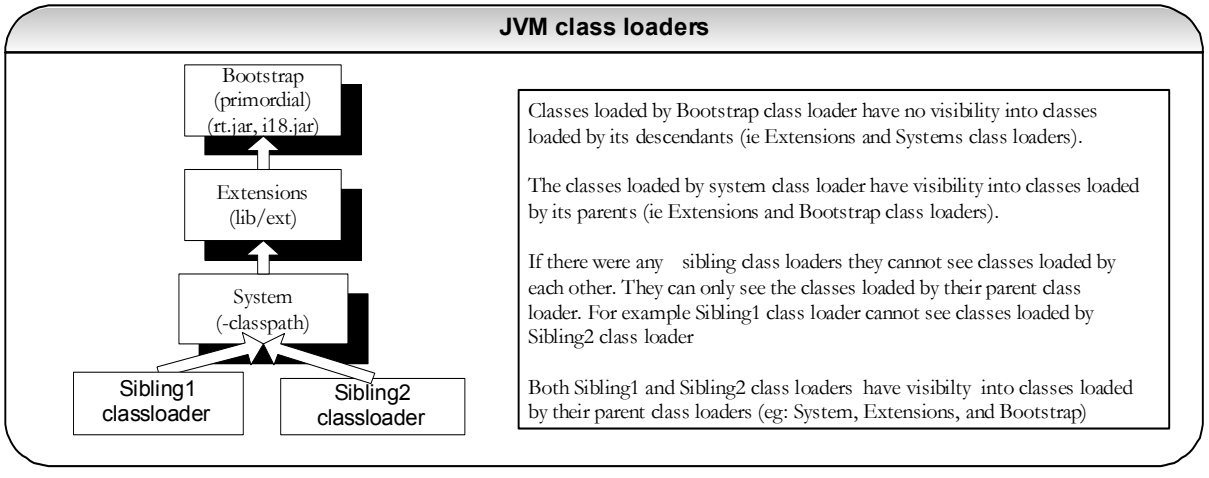
You can create a package as follows with **package** keyword, which is the first keyword in any Java program followed by **import** statements. The **java.lang** package is imported implicitly by default and all the other packages must be explicitly imported.

```
package com.xyz.client ;
import  java.io.File;
import  java.net.URL;
```

Q 05: Explain Java class loaders? If you have a class in a package, what do you need to do to run it? Explain dynamic class loading? **LF**

A 05: 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 especially 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.

Q. What do you need to do to run a class with a main() method in a package?

Example: Say, you have a class named "Pet" in a project folder "c:\myProject" and package named com.xyz.client, will you be able to compile and run it as it is?

```
package com.xyz.client;

public class Pet {
    public static void main(String[] args) {
         System.out.println("I am found in the classpath");
    }
}
```

To run → c:\myProject> `java com.xyz.client.Pet`

The answer is no and you will get the following exception: "Exception in thread "main" java.lang.-NoClassDefFoundError: com/xyz/client/Pet". You need to set the classpath. How can you do that? One of the following ways:

1. Set the operating system **CLASSPATH** environment variable to have the project folder "c:\myProject". [Shown in the above diagram as the System –classpath class loader]
2. Set the operating system **CLASSPATH** environment variable to have a jar file "c:/myProject/client.jar", which has the *Pet.class* file in it. [Shown in the above diagram as the System –classpath class loader].
3. Run it with –cp or –classpath option as shown below:

```
c:\>java -cp  c:/myProject  com.xyz.client.Pet
              OR
c:\>java -classpath c:/myProject/client.jar  com.xyz.client.Pet
```

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 **Q51** in Java section for singleton design pattern]

Q. 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 a Constants class. **// 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***.

Q. 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 StaticInitializer {
   public static final int A = 5;
   public static final int B; //note that it is not → public static final int B = null;
   //note that since B is final, it can be initialized only once.

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

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

   public StaticInitializer(){}  //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 =" + StaticInitializer.A + ", B =" + StaticInitializer.B);
}
```

Q 06: What is the difference between constructors and other regular methods? What happens if you do not provide a constructor? Can you call one constructor from another? How do you call the superclass's constructor? **LF** **FAQ**

A 06:

Constructors	Regular methods
Constructors must have the same name as the class name and cannot return a value. The constructors are called only once per creation of an object while regular methods can be called many times. E.g. for a Pet.class `public Pet() {}  // constructor`	Regular methods can have any name and can be called any number of times. E.g. for a Pet.class. `public void Pet(){}  // regular method has a void return type.` **Note:** method name is shown starting with an uppercase to differentiate a constructor from a regular method. Better naming convention is to have a meaningful name starting with a lowercase like: `public void createPet(){}  // regular method has a void return type`

Q. What happens if you do not provide a constructor? Java does not actually require an explicit constructor in the class description. If you do not include a constructor, the Java compiler will create a default constructor in the byte code with an empty argument. This default constructor is equivalent to the explicit "Pet(){}". If a class includes one or more explicit constructors like "public Pet(int id)" or "Pet(){}" etc, the java compiler does *not* create the default constructor "Pet(){}".

Q. Can you call one constructor from another? Yes, by using **this()** syntax. E.g.

```
public Pet(int id) {
    this.id = id;                       // "this" means this object
}
public Pet (int id, String type) {
    this(id);                           // calls constructor public Pet(int id)
    this.type = type;                  // "this" means this object
}
```

Q. How to call the superclass constructor? If a class called "*SpecialPet"* extends your "*Pet"* class then you can use the keyword "**super**" to invoke the superclass's constructor. E.g.

```
public SpecialPet(int id) {
     super(id);                        //must be the very first statement in the constructor.
}
```

To call a regular method in the super class use: "**super**.myMethod( );". This can be called at any line. Some frameworks based on JUnit add their own initialization code, and not only do they need to remember to invoke

their parent's setup() method, you, as a user, need to remember to invoke theirs after you wrote your initialization code:

```
public class DBUnitTestCase extends TestCase {
  public void setUp() {
    super.setUp();
    // do my own initialization
  }
}
```

```
public void cleanUp() throws Throwable
{
     try {
       … // Do stuff here to clean up your object(s).
     }
     catch (Throwable t) {}
     finally{
         super.cleanUp(); //clean up your parent class. Unlike constructors
                          // super.regularMethod() can be called at any line.
     }
}
```

Q 07: What are the advantages of Object Oriented Programming Languages (OOPL)? DC FAQ

A 07: 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 08: How does the Object Oriented approach improve software development? DC

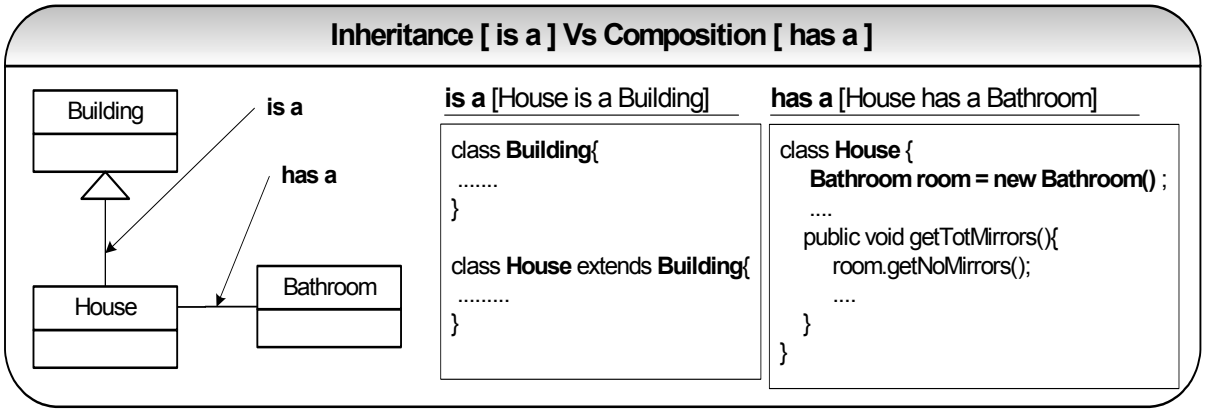
A 08: The key benefits are:

- ***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 09: 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 FAQ

A 09: 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.

Q. Which one to favor, composition or inheritance? 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 **Q10** 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 is deleted then all corresponding line items for that order should be deleted. So **composition has a stronger relationship**.

Q 10: What do you mean by polymorphism, inheritance, encapsulation, and dynamic binding? **DC** **SE** **FAQ**

A 10: ***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:

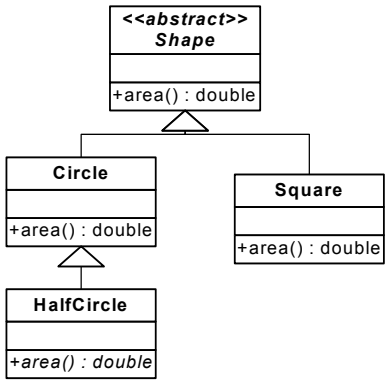
```
//client or calling code
double dim = 5.0; //ie 5 meters radius or width
List listShapes = new ArrayList(20);

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 ;
}
```

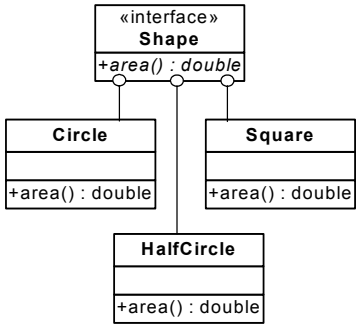
For example: given a base class/interface *Shape*, polymorphism allows the programmer to define different area(double dim1) methods for any number of derived classes such as *Circle*, *Square* etc. No matter what shape an object is, applying the area method to it will return the right results.

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 behavior (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 application's 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 behavior of the superclass, not on the actual implementation**. For example in the above diagram, the subclasses should only be concerned about the behavior 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 favors interface inheritance to implementation inheritance. **CO**

Implementation inheritance

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

Super class *Account* has **reusable code** as methods **deposit** (double amount) and **withdraw** (double amount).

```
public abstract class Account  {
   public void deposit (double amount) {
        System.out.println("depositing " + amount);
   }

   public void withdraw (double amount) {
        System.out.println ("withdrawing " + amount);
   }

   public abstract double calculateInterest(double  amount);
}
```

```
public class SavingsAccount extends Account {

    public double calculateInterest (double amount) {
       // calculate interest for SavingsAccount
       return amount * 0.03;
    }

    public void deposit (double amount) {
       super.deposit (amount); // get code reuse
       // do something else
    }

    public void withdraw (double amount) {
```

Interface inheritance with composition

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 double calculateInterest(double amount);
    public abstract void deposit(double amount);
    public abstract void withdraw(double amount);
}
```

Code to interface so that the implementation can change.

```
public interface AccountHelper {
    public abstract void deposit (double amount);
    public abstract void withdraw (double amount);
}
```

class **AccountHelperImpl** has **reusable code** as methods **deposit** (double amount) and **withdraw** (double amount).

```
public class AccountHelperImpl implements AccountHelper {
   public void deposit(double amount) {
        System.out.println("depositing " + amount);
   }

   public void withdraw(double amount) {
        System.out.println("withdrawing " + amount);
   }

}
```

```
public class SavingsAccountImpl implements Account {
```

```
        super.withdraw (amount); // get code reuse
        // do something else
    }
}
```

```
public class TermDepositAccount extends Account {

    public double calculateInterest (double amount) {
        // calculate interest for SavingsAccount
        return amount * 0.05;
    }

    public void deposit(double amount) {
        super.deposit (amount); // get code reuse
        // do something else
    }

    public void withdraw(double amount) {
        super.withdraw (amount); // get code reuse
        // do something else
    }
}
```

```
    // composed helper class (i.e. composition).
    AccountHelper helper = new AccountHelperImpl ();

    public double calculateInterest (double amount) {
        // calculate interest for SavingsAccount
        return amount * 0.03;
    }

    public void deposit (double amount) {
        helper.deposit( amount); // code reuse via composition
    }

    public void withdraw (double amount) {
        helper.withdraw (amount); // code reuse via composition
    }
}
```

```
public class TermDepositAccountImpl implements Account {

    // composed helper class (i.e. composition).
    AccountHelper helper = new AccountHelperImpl  ();

    public double calculateInterest (double amount) {
        //calculate interest for SavingsAccount
        return amount * 0.05;
    }

    public void deposit (double amount) {
         helper.deposit (amount) ;  // code reuse via composition
    }

    public void withdraw (double amount) {
          helper.withdraw (amount) ;  // code reuse via composition
    }

}
```

The Test class:

```
public class Test {
    public static void main(String[] args) {
        Account acc1 = new SavingsAccountImpl();
        acc1.deposit(50.0);

        Account acc2 = new TermDepositAccountImpl();
        acc2.deposit(25.0);

        acc1.withdraw(25);
        acc2.withdraw(10);

        double cal1 = acc1.calculateInterest(100.0);
        double cal2 = acc2.calculateInterest(100.0);

        System.out.println("Savings --> " + cal1);
        System.out.println("TermDeposit -->  " + cal2);
     }
}
```

The output:

```
depositing 50.0
depositing 25.0
withdrawing 25.0
withdrawing 10.0
Savings --> 3.0
TermDeposit -->  5.0
```

Q. Why would you prefer code reuse via composition over inheritance? Both the approaches make use of polymorphism and gives code reuse (in different ways) to achieve the same results but:

- The advantage of class inheritance is that it is done statically at compile-time and is easy to use. The disadvantage of class inheritance is that because it is static, implementation inherited from a parent class cannot be changed at run-

time. In object composition, functionality is acquired dynamically at run-time by objects collecting references to other objects. The advantage of this approach is that implementations can be replaced at run-time. This is possible because objects are accessed only through their interfaces, so one object can be replaced with another just as long as they have the same type. **For example:** the composed class ***AccountHelperImpl*** can be replaced by another more efficient implementation as shown below if required:

```
public class EfficientAccountHelperImpl implements AccountHelper {
    public void deposit(double amount) {
        System.out.println("efficient depositing " + amount);
    }

    public void withdraw(double amount) {
        System.out.println("efficient withdrawing " + amount);
    }
}
```

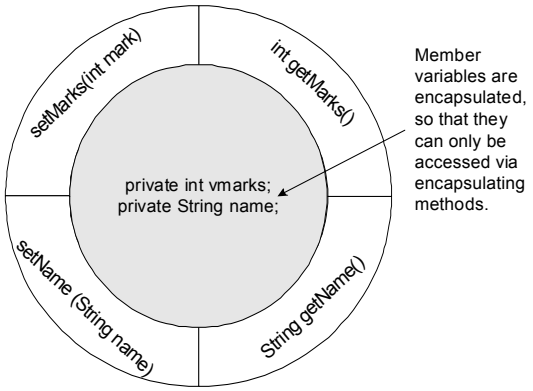
- Another problem with class inheritance is that the subclass becomes dependent on the parent class implementation. This makes it harder to reuse the subclass, especially if part of the inherited implementation is no longer desirable and hence can break encapsulation. Also a change to a superclass can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclasses making the design fragile by tightly coupling the subclasses with the super class. But it is easier to change the interface/implementation of the composed class.

Due to the flexibility and power of object composition, **most design patterns emphasize object composition over inheritance whenever it is possible**. Many times, a design pattern shows a clever way of solving a common problem through the use of object composition rather then a standard, less flexible, inheritance based solution.

Encapsulation – refers to keeping all the related members (variables and methods) together in an object. Specifying member variables 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. **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.
}
```



Being able to encapsulate members of a class is important for **security** and **integrity**. We can protect variables from unacceptable values. The sample code above 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.

Q 11: What is design by contract? Explain the *assertion* construct? **DC**

A 11: 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 **Q39** 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, "pilotJob" never gets 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; //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 error's 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
```

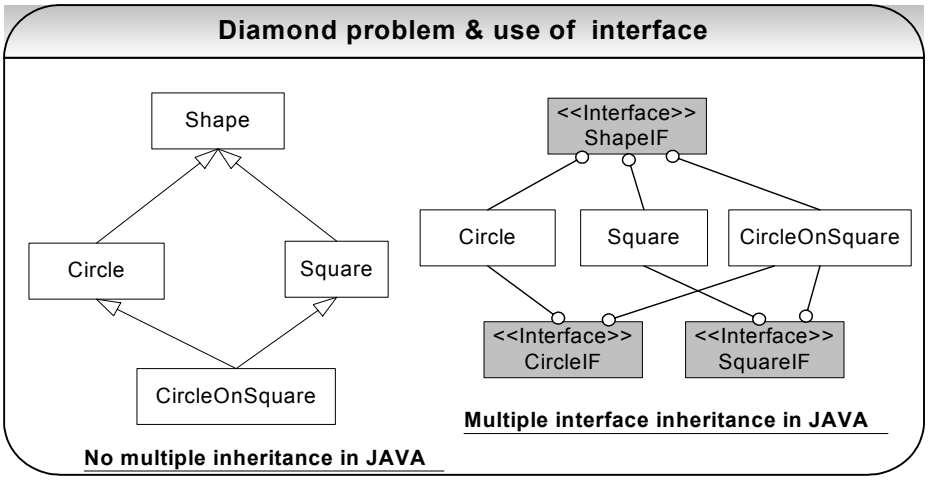
Q 12: What is the difference between an abstract class and an interface and when should you use them? **LF** **DP** **DC** **FAQ**

A 12: 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 behavior), 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.



Q. 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 behavior and force subclasses to provide any specific behavior. Care should be taken not to overuse implementation inheritance as discussed in **Q10** in Java section.

Q. 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**. **For example:** The Spring framework's dependency injection promotes code to an interface principle. 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 inheritance. 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 13: Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java? **LF** **FAQ**

A 13: 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*** java.io.Serializable (Refer **Q23** in Java section), java.lang.Cloneable, java.util.EventListener etc. Marker interfaces are also known as "tag" interfaces since they tag all the derived classes into a category based on their purpose.

Q 14: When is a method said to be overloaded and when is a method said to be overridden? **LF** **CO** **FAQ**

A 14:

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 15: What is the main difference between an ArrayList and a Vector? What is the main difference between HashMap and Hashtable? What is the difference between a stack and a queue? **LF** **DC** **PI** **CI** **FAQ**

A 15:

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.

Q. 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*ollections* API's external synchronization facility as shown below to **temporarily synchronize** your collections as needed: **CO**

```
Map myMap = Collections.synchronizedMap (myMap);          // single lock for the entire map
List  myList = Collections.synchronizedList (myList);     // single lock for the entire list
```

J2SE 5.0: If you are using J2SE5, you should use the new *"**java.util.concurrent**"* package for improved performance because the concurrent package collections are not governed by a single synchronized lock as shown above. The "*java.util.concurrent"* package collections like **ConcurrentHashMap** is threadsafe and at the same time safely permits any number of concurrent reads as well as tunable number of concurrent writes. The "java.util.concurrent" package also provides an efficient scalable thread-safe non-blocking FIFO queue like **ConcurrentLinkedQueue**.

J2SE 5.0: The "*java.util.concurrent"* package also has classes like **CopyOnWriteArrayList**, **CopyOnWrite-ArraySet**, which gives you thread safety with the added benefit of immutability to deal with data that changes infrequently. The **CopyOnWriteArrayList** behaves much like the ArrayList class, except that when the list is modified, instead of modifying the underlying array, a new array is created and the old array is discarded. This means that when a caller gets an iterator (i.e. `copyOnWriteArrayListRef.iterator()`), which internally holds a reference to the underlying CopyOnWriteArrayList object's array, which is immutable and therefore can be used for traversal without requiring either synchronization on the list `copyOnWriteArrayListRef` or need to clone() the `copyOnWriteArrayListRef` list before traversal (i.e. there is no risk of concurrent modification) and also offers better performance.

Array	List / Stack etc
Java arrays are even faster than using an *ArrayList/Vector* and perhaps therefore may be preferable if you know the size of your array upfront (because **arrays cannot grow as Lists do**).	*ArrayList/Vector* are specialized data structures that internally uses an array with some convenient methods like add(..), remove(…) etc so that they can grow and shrink from their initial size. ArrayList also supports index based searches with indexOf(Object obj) and lastIndexOf(Object obj) methods.
In an array, any item can be accessed.	These are more abstract than arrays and access is restricted. For example, a stack allows access to only last item inserted.

Queue<E> (added in J2SE 5.0)	Stack
First item to be inserted is the first one to be removed.	Allows access to only last item inserted.
This mechanism is called First In First Out (FIFO).	An item is inserted or removed from one end called the "top" of the stack. This is called Last In First Out (**LIFO**) mechanism.
Placing an item in the queue is called "enqueue or insertion" and removing an item from a queue is called "dequeue or deletion". Pre J2SE 5.0, you should write your own *Queue* class with enqueue() and dequeue() methods using an *ArrayList* or a *LinkedList* class. J2SE 5.0 has **a java.util.Queue<E>** interface.	Placing the data at the top is called "pushing" and removing an item from the top is called "popping". If you want to reverse "XYZ" → ZYX, then you can use a **java.util.Stack**

Q 16: Explain the Java Collections Framework? **LF** **DP** **FAQ**

A 16: The key interfaces used by the collections 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.

Set (HashSet , TreeSet)	List (ArrayList, LinkedList, Vector etc)
A ***Set*** is a collection with **unique elements** and prevents duplication within the collection. ***HashSet*** and ***TreeSet*** are implementations of a *Set* interface. A TreeSet is an ordered HashSet, which implements the **SortedSet** 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. (i.e. an index based)

The Collections 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. A **TreeMap** is an ordered HashMap, which implements the **SortedMap** interface.

Q. How to implement collection ordering? ***SortedSet*** and ***SortedMap*** interfaces maintain sorted order. The classes, which implement the ***Comparable*** interface, impose natural order. By implementing **Comparable**, sorting an array of objects or a collection (List etc) is as simple as:

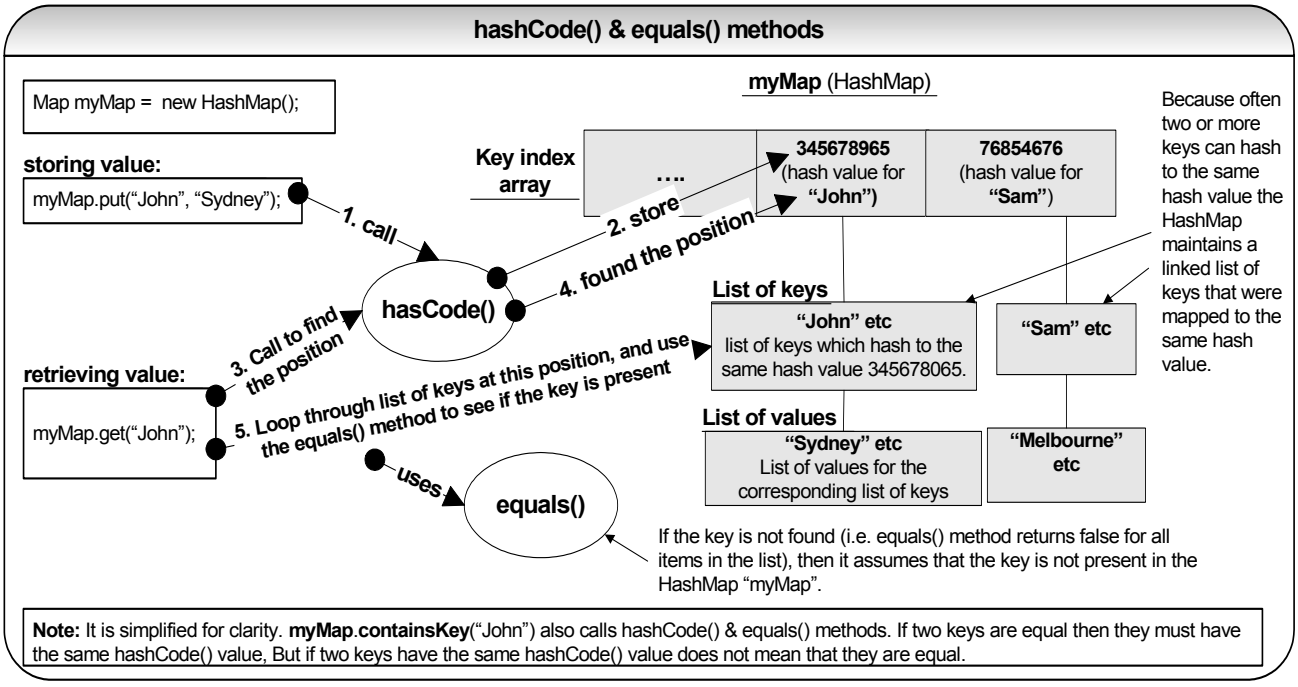
```
Arrays.sort(myArray);
Collections.sort(myCollection);      // do not confuse "Collections" utility class with the
                                     // "Collection" interface without an "s".
```

method with public access modifier	followed by an "@" sign and then unsigned hexadecimal representation of the hashcode, for example Pet@162b91. This hexadecimal representation is not what the users of your class want to see. Providing your toString() method makes your class much more pleasant to use and it is recommended that all subclasses override this method. The toString() method is invoked automatically when your object is passed to println(), assert() or the string concatenation operator (+). `public class Pet {` `  int id;` `  String name;` `  public boolean equals(Object obj){` `  //as shown above.` `  }` `  public int hashCode() {` `  //as shown before` `  }` `  public String toString() {` `  StringBuffer sb = new StringBuffer();` `  sb.append("id=").append(id);` `  sb.append(",name=").append(name);` `  return sb.toString();` `  }` `}`
clone() method with protected access modifier	You should override the clone() method very judiciously. Implementing a properly functioning clone method is complex and it is rarely necessary. You are better off providing some alternative means of object copying (refer **Q26** in Java section) or simply not providing the capability. A better approach is to provide a copy constructor or a static factory method in place of a constructor. //constructor `public Pet(Pet petToCopy){` `  …` `}` //static factory method `public static Pet newInstance(Pet petToCopy){` `  …` `}` The clone() method can be disabled as follows: `public final Object clone() throws CloneNotSupportedException {` `  throw new CloneNotSupportedException();` `}`
finalize() method with protected access modifier	Unlike C++ destructors, the finalize() method in Java is unpredictable, often dangerous and generally unnecessary. Use try{} finally{} blocks as discussed in **Q32** in Java section & **Q45** in Enterprise section. The finalize() method should only be used in rare instances as a safety net or to terminate non-critical native resources. If you do happen to call the finalize() method in some rare instances then remember to call the super.finalize() as shown below: `protected void finalize() throws Throwable {` `  try{` `  //finalize subclass state` `  }` `  finally {` `  super.finalize();` `  }` `}`

Q 20: 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 FAQ

A 20: 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 color 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.



Q. What are the primary considerations when implementing a user defined key?

- 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.
- It is a best practice to implement the user defined key class as an immutable (refer **Q21**) object.

Q. Why it is a best practice to implement the user defined key class as an immutable object?

Problem: As per the code snippet shown below if you use a mutable user defined class "**UserKey**" as a HashMap key and subsequently if you mutate (i.e. modify via setter method e.g. `key.setName("Sam")`) the key after the object has been added to the HashMap then you will not be able to access the object later on. The original key object will still be in the HashMap (i.e. you can iterate through your HashMap and print it – both prints as "Sam" as opposed to "John" & Sam) but you cannot access it with `map.get(key)` or querying it with `map.containsKey(key)` will return false because the key "John" becomes "Sam" in the "List of keys" at the key index "345678965" if you mutate the key after adding. These types of errors are very hard to trace and fix.

```
Map myMap = new HashMap(10);
//add the key "John"
UserKey key = new UserKey("John");  //Assume UserKey class is mutable
myMap.put(key, "Sydney");
//now to add the key "Sam"
key.setName("Sam"); // same key object is mutated instead of creating a new instance.
                    // This line modifies the key value "John" to "Sam" in the "List of keys"
                    // as shown in the diagram above. This means that the key "John" cannot be
                    // accessed. There will be two keys with "Sam" in positions with hash
                    // values 345678965 and 76854676.
myMap.put(key, "Melbourne");

myMap.get(new UserKey("John")); // key cannot be accessed. The key hashes to the same position
             // 345678965 in the "Key index array" but cannot be found in the "List of keys"
```

Solution: Generally you use a java.lang.Integer or a java.lang.String class as the key, which are immutable Java objects. If you define your own key class then it is a best practice to make the key class an immutable object (i.e. do not provide any setXXX() methods in your key class. e.g. no `setName(…)` method in the ***UserKey*** class). If a programmer wants to insert a new key then he/she will always have to instantiate a new object (i.e. cannot mutate the existing key because immutable key object class has no setter methods).

```
Map myMap = new HashMap(10);
//add the key "John"
UserKey key1 = new UserKey("John");  //Assume UserKey is immutable
myMap.put(key1, "Sydney");
```

```
//add the key "Sam"
UserKey key2 = new UserKey("Sam");  //Since UserKey is immutable, new instance is created.
myMap.put(key2, "Melbourne");

myMap.get(new UserKey("John"));     //Now the key can be accessed
```

Similar issues are possible with the *Set* (e.g. HashSet) as well. If you add an object to a "Set" and subsequently modify the added object and later on try to query the original object it may not be present. `mySet.contains(originalObject)` may return false.

J2SE 5.0 introduces enumerated constants, which improves readability and maintainability of your code. Java programming language enums are more powerful than their counterparts in other languages. **Example:** As shown below 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()) { //using J2SE 5.0 for each loop
                 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 21: What is the main difference between a String and a StringBuffer class? **LF PI CI CO FAQ**

A 21:

String	StringBuffer / StringBuilder (added in J2SE 5.0)
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 synchronized, 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**);// set an initial size of 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 initialize 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 operator "+" 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. What is an immutable object? Immutable objects whose state (i.e. the object's data) does not change once it is instantiated (i.e. it becomes a read-only object after instantiation). Immutable classes are ideal for representing

numbers (e.g. java.lang.Integer, java.lang.Float, java.lang.BigDecimal etc are immutable objects), enumerated types, colors (e.g. java.awt.Color is an immutable object), short lived objects like events, messages etc.

Q. What are the benefits of immutable objects?

- Immutable classes can greatly simplify programming by freely allowing you to cache and share the references to the immutable objects without having to defensively copy them or without having to worry about their values becoming stale or corrupted.
- Immutable classes are inherently thread-safe and you do not have to synchronize access to them to be used in a multi-threaded environment. So there is no chance of negative performance consequences.
- Eliminates the possibility of data becoming inaccessible when used as keys in HashMaps or as elements in Sets. These types of errors are hard to debug and fix. Refer **Q20** in Java section under "**Q. Why it is a best practice to implement the user defined key class as an immutable object?** "

Q. How will you write an immutable class? CO

Writing an immutable class is generally easy but there can be some tricky situations. Follow the following guidelines:

1. A class is declared final (i.e. final classes cannot be extended).

```
public final class MyImmutable { … }
```

2. All its fields are final (final fields cannot be mutated once assigned).

```
private final int[] myArray; //do not declare as → private final int[] myArray = null;
```

3. Do not provide any methods that can change the state of the immutable object in any way – not just setXXX methods, but any methods which can change the state.
4. The "this" reference is not allowed to escape during construction from the immutable class and the immutable class should have exclusive access to fields that contain references to mutable objects like arrays, collections and mutable classes like Date etc by:
   - Declaring the mutable references as private.
   - Not returning or exposing the mutable references to the caller (this can be done by defensive copying)

Wrong way to write an immutable class

Wrong way to write a constructor:

```
public final class MyImmutable {

  private final int[] myArray;

  public MyImmutable(int[] anArray) {
      this.myArray = anArray; // wrong
  }

  public String toString() {
     StringBuffer sb = new StringBuffer("Numbers are: ");
     for (int i = 0; i < myArray.length; i++) {
         sb.append(myArray[i] + " ");
    }
    return sb.toString();
  }
}
```

// the caller could change the array after calling the constructor.

```
int[] array = {1,2};
MyImmutable myImmutableRef = new MyImmutable(array) ;
System.out.println("Before constructing " + myImmutableRef);
array[1] = 5; // change (i.e. mutate)  the element
System.out.println("After constructing " + myImmutableRef);
```

Out put:

```
Before constructing Numbers are: 1 2
```

Right way to write an immutable class

Right way is to copy the array before assigning in the constructor.

```
public final class MyImmutable {

  private final int[] myArray;

  public MyImmutable(int[] anArray) {
      this.myArray = anArray.clone();  // defensive copy
  }

  public String toString() {
     StringBuffer sb = new StringBuffer("Numbers are: ");
     for (int i = 0; i < myArray.length; i++) {
         sb.append(myArray[i] + " ");
    }
    return sb.toString();
  }
}
```

// the caller cannot change the array after calling the constructor.

```
int[] array = {1,2};
MyImmutable myImmutableRef = new MyImmutable(array) ;
System.out.println("Before constructing " + myImmutableRef);
array[1] = 5; // change (i.e. mutate)  the element
System.out.println("After constructing " + myImmutableRef);
```

Out put:

```
Before constructing Numbers are: 1 2
```

`After constructing Numbers are: 1 5` As you can see in the output that the "MyImmutable" object has been mutated. This is because the object reference gets copied as discussed in **Q22** in Java section.	After constructing Numbers are: 1 2 As you can see in the output that the "MyImmutable" object has not been mutated.
Wrong way to write an accessor. A caller could get the array reference and then change the contents: `public int[] getArray() {` `  return myArray;` `}`	Right way to write an accessor by cloning. `public int[] getAray() {` `  return (int[]) myArray.clone();` `}`

Important: Beware of using the clone() method on a collection like a Map, List, Set etc because they are not only difficult to implement correctly refer **Q19** in Java section but also the default behavior of an object's clone() method automatically yields a shallow copy. You have to deep copy the mutable objects referenced by your immutable class. Refer **Q26** in Java section for deep vs. shallow cloning and **Q22** in Java section for why you will be modifying the original object if you do not deep copy.

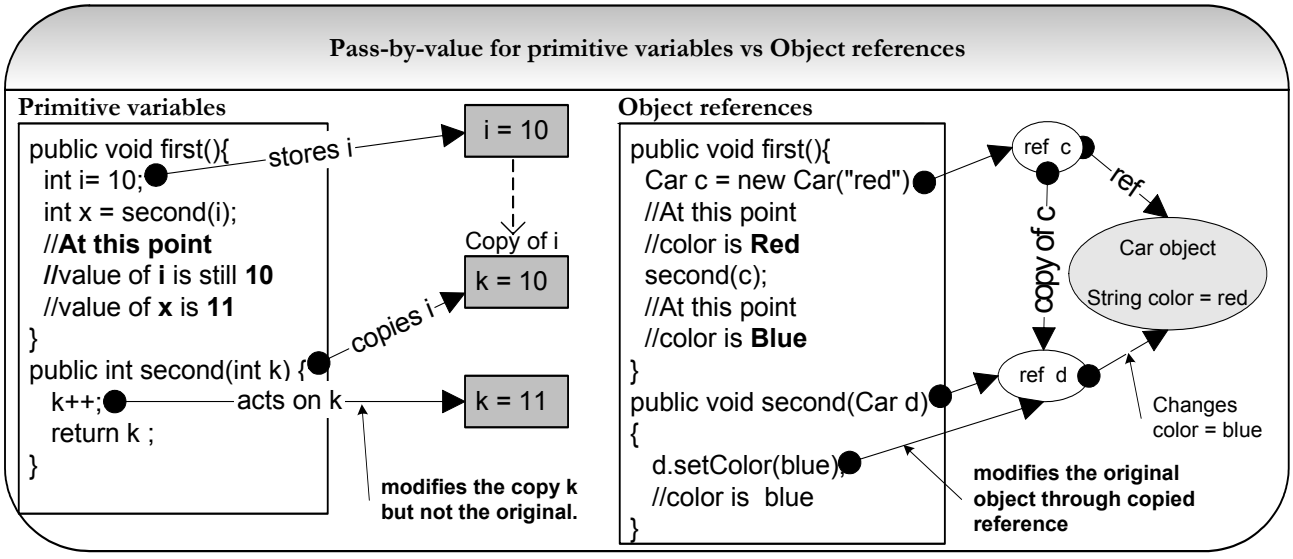
Q. How would you defensively copy a Date field in your immutable class?

```
public final class MyDiary {

    private Date myDate = null;

    public MyDiary(Date aDate){
        this.myDate = new Date(aDate.getTime());       // defensive copying by not exposing the "myDate" reference
    }

     public Date getDate() {
          return new Date(myDate.getTime);             // defensive copying by not exposing the "myDate" reference
     }
}
```

Q 22: What is the main difference between pass-by-reference and pass-by-value? LF PI FAQ

A 22: 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

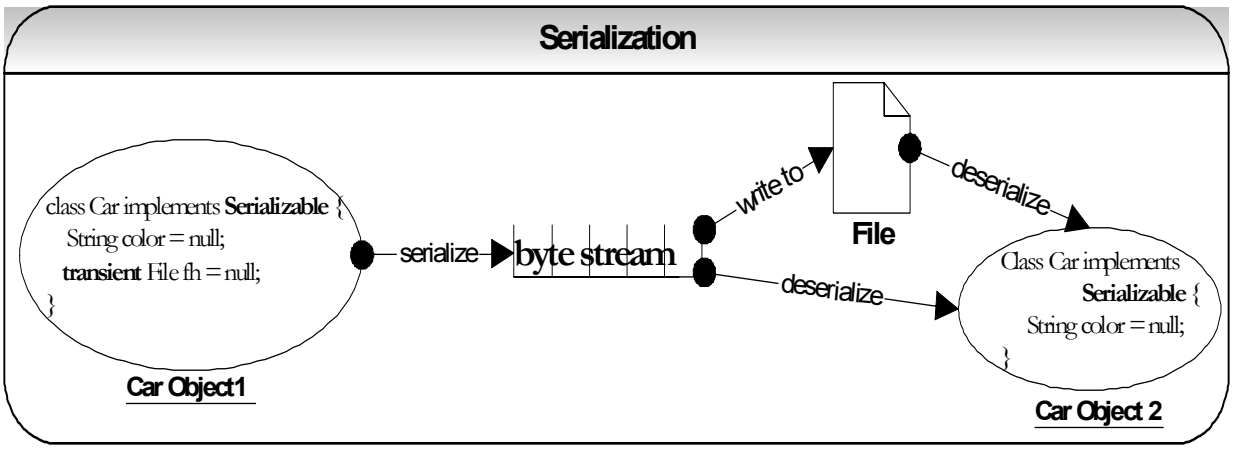
JVM). Hence inter-process communication involves calling method passing objects as arguments to called method **by-value** in a serialized form, which can adversely affect performance due to marshaling and unmarshaling cost.

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 23: 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? What is a serial version id? LF SI PI FAQ

A 23: 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, a database connection, a system thread etc. 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.

Q. 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 **Q26** in Java section).

To serialize the above "*Car*" object to a file (sample for illustration purpose only, should use try {} catch {} block):

```
Car car = new Car();   // The "Car" class implements a java.io.Serializable interface
FileOutputStream fos = new FileOutputStream(filename);
ObjectOutputStream out = new  ObjectOutputStream(fos);
out.writeObject(car); // serialization mechanism happens here
out.close();
```

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. What is Java Serial Version ID? Say you create a "Car" class, instantiate it, and write it out to an object stream. The flattened car object sits in the file system for some time. Meanwhile, if the "Car" class is modified by adding a new field. Later on, when you try to read (i.e. deserialize) the flattened "Car" object, you get the **java.io.InvalidClassException** – because all serializable classes are automatically given a unique identifier. This exception is thrown when the identifier of the class is not equal to the identifier of the flattened object. If you really think about it, the exception is thrown because of the addition of the new field. You can avoid this exception being thrown by controlling the versioning yourself by declaring an explicit **serialVersionUID**. There is also a small

performance benefit in explicitly declaring your serialVersionUID (because does not have to be calculated). So, it is best practice to add your own **serialVersionUID** to your Serializable classes as soon as you create them as shown below:

```
public class Car {
     static final long serialVersionUID = 1L; //assign a long value
}
```

Note: Alternatively you can use the serialver tool comes with Sun's JDK. This tool takes a full class name on the command line and returns the serialVersionUID for that compiled class. **For example:**

```
static final long serialVersionUID = 10275439472837494L; //generated by serialver tool.
```

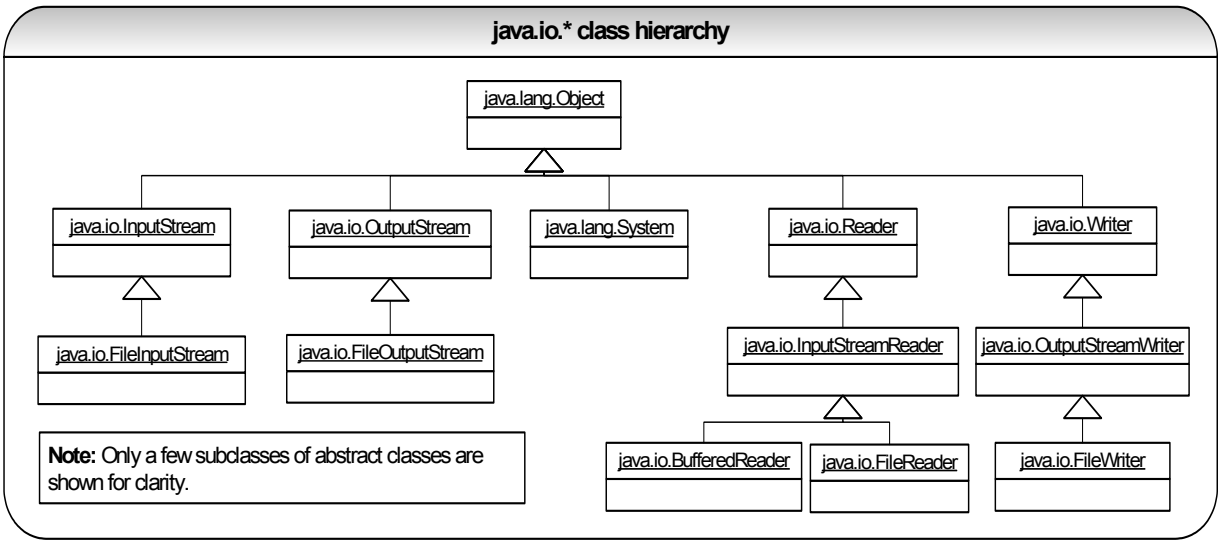
Q 24: Explain the Java I/O streaming concept and the use of the decorator design pattern in Java I/O? **LF** **DP** **PI** **SI**

A 24: 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 behavior at runtime based on some basic classes.

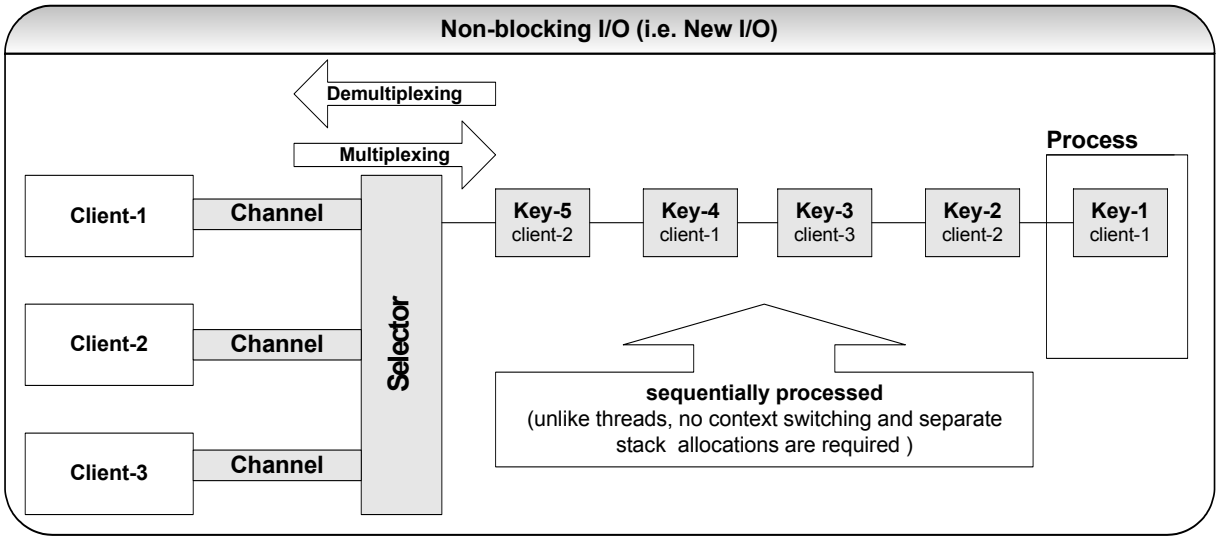
Attaching responsibilities to classes at compile time using subclassing.	Attaching responsibilities to objects at runtime using a decorator design pattern.
Inheritance (aka subclassing) attaches responsibilities to classes at compile time. When you extend a class, each individual changes you make to 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.



Q. How does the new I/O (NIO) offer better scalability 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** (i.e. each thread has its own stack. Refer **Q34**, **Q42** in Java section) and **CPU context switching** (i.e. switching between threads as opposed to doing real computation.). 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 *InputStream* 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.

A **Selector** class enables multiplexing (combining multiple streams into a single stream) and demultiplexing (separating a single stream into multiple streams) I/O events and makes it possible for a single thread to efficiently manage many I/O channels. A Selector monitors selectable channels, which are registered with it for I/O events like connect, accept, read and write. The keys (i.e. Key1, Key2 etc represented by the SelectionKey class) encapsulate the relationship between a specific selectable channel and a specific selector.

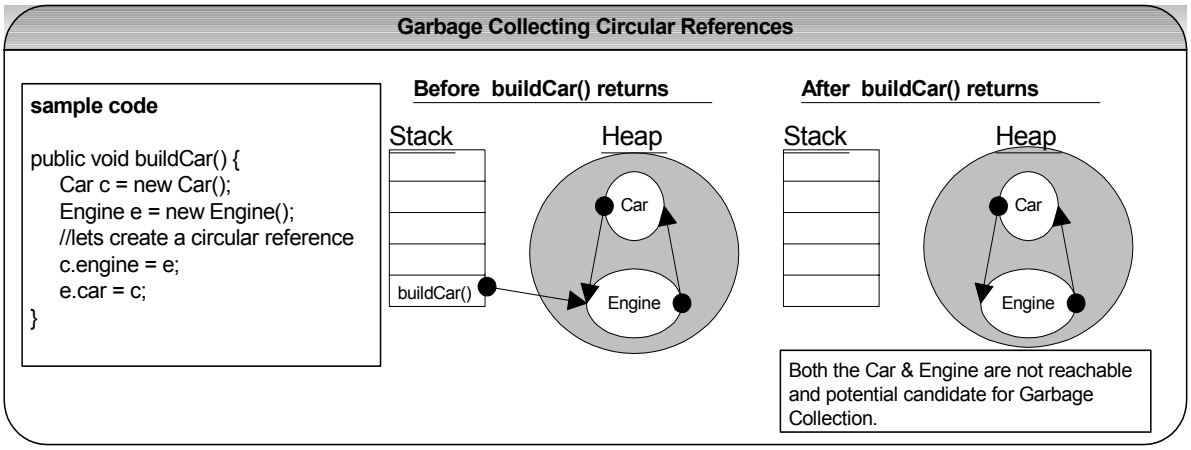
Buffers hold data. Channels can fill and drain *Buffers*. Buffers replace the need for you to do your own buffer management using byte arrays. There are different types of Buffers like *ByteBuffer*, *CharBuffer*, *DoubleBuffer*, etc.

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.

Q 38: 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 38: Yes. Refer diagram below.

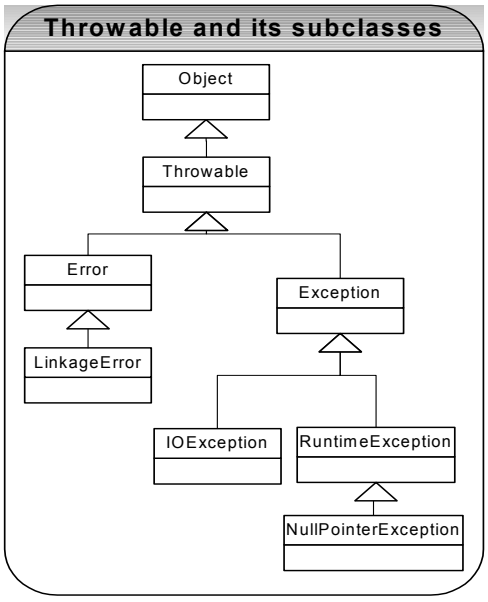


Q 39: Discuss the Java error handling mechanism? What is the difference between Runtime (**unchecked**) exceptions and **checked** exceptions? What is the implication of catching all the exceptions with the type "*Exception*"? EH BP FAQ

A 39:

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.



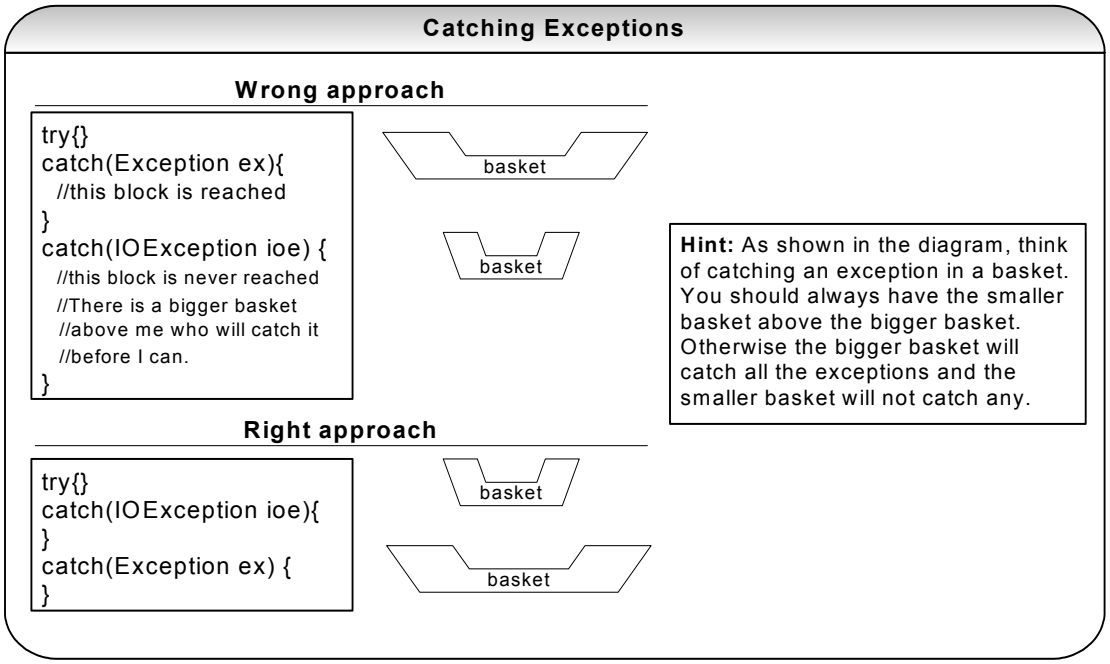
Runtime Exceptions (unchecked exception)

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.

Q. What are the exception handling best practices: BP

1. Q. 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 above 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*, *ArrayIndexOutOfBoundsException*) as well.

Example: The *FileNotFoundException* is extended (i.e. inherited) from the *IOException*. So (subclasses have to be caught first) *FileNotFoundException* (small basket) should be caught before *IOException* (big basket).

2. Q. Why should you throw an exception early? CO

The exception stack trace helps you pinpoint where an exception occurred by showing you 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:

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

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);
…
```

3. 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. It is a bad practice to sweep the exceptions under the carpet by catching it and not doing anything with it. 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.

4. Q. When should you use a checked exception and when should you use an unchecked exception?

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. Convert your SQLException into an unchecked (i.e. RuntimeException) exception, if the client code can not recover from it. (**Note:** Hibernate 3 & Spring uses RuntimeExceptions prevalently).

Important: throw an exception early and catch an exception late but do not sweep an exception under the carpet by catching it and not doing anything with it. This will hide problems and it will be hard to debug and fix. **CO**

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 40: What is a user defined exception? **EH**
A 40: 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. To throw an exception:

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

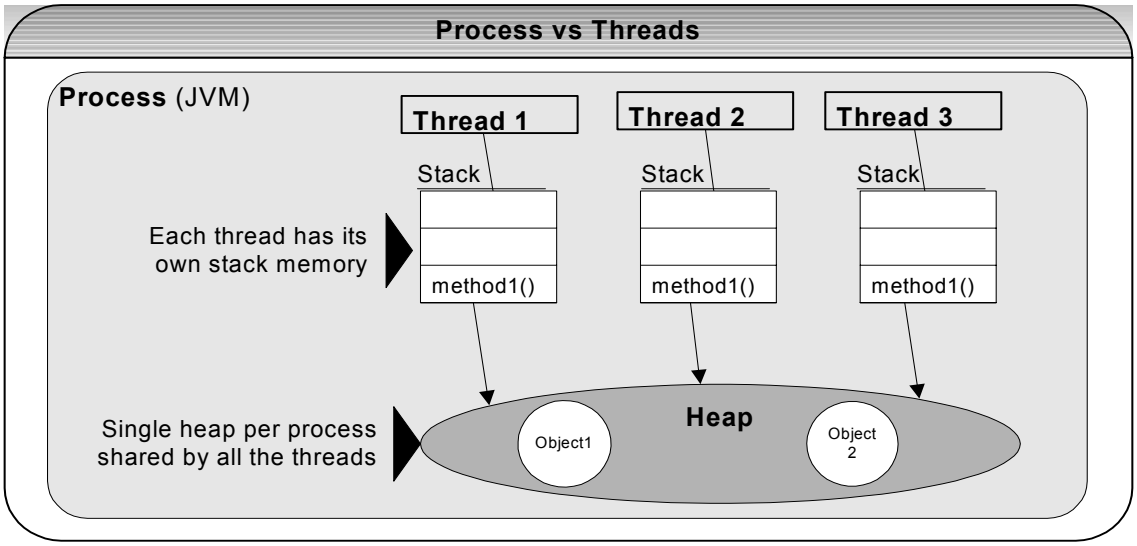
To declare an exception: `public myMethod() throws MyException {…}`

Q 41: What are the flow control statements in Java? **LF**
A 41: The flow control statements allow you to conditionally execute statements, to repeatedly execute a block of statements, or to just change the sequential flow of control.

Flow control types	Keyword
Looping	**while**, **do-while**, **for** The body of the while loop is executed only if the expression is true, so it may not be executed even once: `while(i < 5){...}` The body of the do-while loop is executed at least once because the test expression is evaluated only after executing the loop body. Also, don't forget the ending semicolon after the while expression. `do { … } while(i < 5);` The for loop syntax is: `for(expr1; expr2; expr3)` `  {` `  // body` `  }` expr1 → is for initialization, expr2 → is the conditional test, and expr3 → is the iteration expression. Any of these three sections can be omitted and the syntax will still be legal: `for( ; ; ) {}  // an endless loop`
Decision making	**if-else**, **switch-case** The if-else statement is used for decision-making -- that is, it decides which course of action needs to be taken. `if (x == 5) {…} else {..}` The switch statement is also used for decision-making, based on an integer expression. The argument passed to the switch and case statements should be **int, short, char,** or **byte**. The argument passed to the case statement should be a literal or a final variable. If no case matches, the default statement (which is optional) is executed. `int i = 1;` `switch(i)` `{` `case 0:` `  System.out.println("Zero");break; //if break; is omitted case 1: also executed` `case 1:` `  System.out.println("One");break; //if break; is omitted default: also executed` `default:` `  System.out.println("Default");break;` `}`
Branching	**break, continue, label:, return** The **break** statement is used to exit from a loop or switch statement, while the **continue** statement is used to skip just the current iteration and continue with the next. The **return** is used to return from a method based on a condition. The **label** statements can lead to unreadable and unmaintainable spaghetti code hence should be avoided.
Exception handling	**try-catch-finally, throw** Exceptions can be used to define ordinary flow control. This is a misuse of the idea of exceptions, which are meant only for exceptional conditions and hence should be avoided.

Q 42: What is the difference between processes and threads? LF MI CI

A 42: 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 43: Explain different ways of creating a thread? LF FAQ

A 43: 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.
    }
}
```

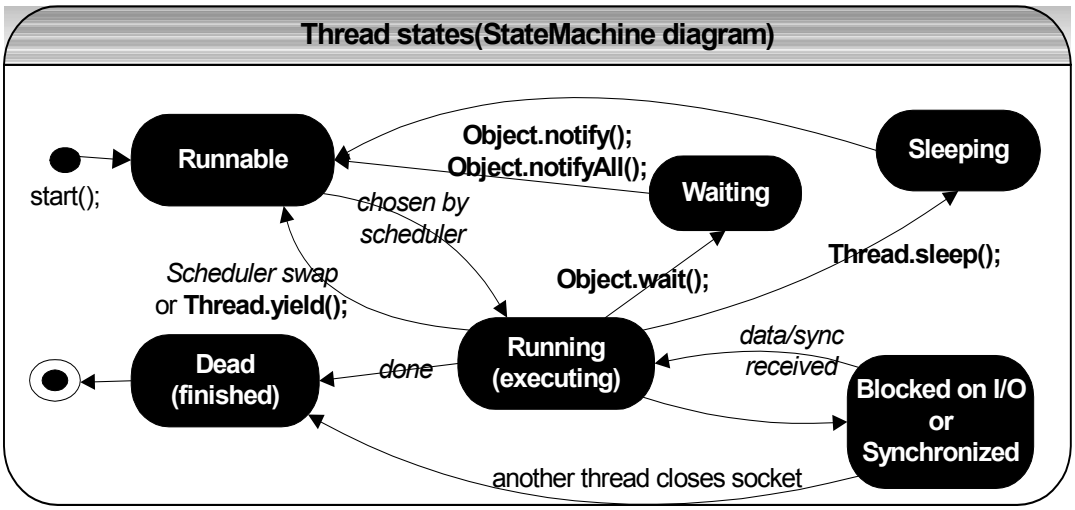
```
class Counter extends Base implements Runnable {

    //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.
    }
}
```

Q. Which one would you prefer and why? 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. In an OO approach you

should only extend a class when you want to make it different from it's superclass, and change it's behavior. By implementing a *Runnable* interface instead of extending the *Thread* class, you are telling to the user that the class *Counter* that an object of type *Counter* will run as a thread.

Q 44: Briefly explain high-level thread states? LF

A 44: 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 scheduler 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 45: What is the difference between yield and sleeping? What is the difference between the methods sleep() and wait()? LF FAQ

A 45: 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.

The method wait(1000), causes the current thread to sleep up to one second. A thread could sleep less than 1 second if it receives the notify() or notifyAll() method call. Refer **Q48** in Java section on thread communication. The call to sleep(1000) causes the current thread to sleep for exactly 1 second.

Q 46: 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 FAQ

A 46: 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
       // because will not be thread-safe

        synchronized(xLock) {
            synchronized(yLock) {
                //access x,y here. thread safe
             }
         }

        //do something here but don't use  SharedResource  x, y
        //because will not be thread-safe
    }//end of method1
}
```

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.

Q. 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 affect the performance.

Q. What is a *ThreadLocal* class? *ThreadLocal* is a handy class for simplifying development of thread-safe concurrent programs by making the object stored in this class not sharable between threads. *ThreadLocal* class encapsulates non-thread-safe classes to be safely used in a multi-threaded environment and also allows you to create per-thread-singleton. **For *ThreadLocal* example:** Refer **Q15 (What is a Session?)** in Emerging Technologies/Frameworks section. Refer **Q51** in Java section for s*ingleton* design pattern.

Q 47: What is a daemon thread? **LF**

A 47: Daemon threads are sometimes called "service" or "background" threads. These are threads that normally run at a low priority and provide a basic service to a program when activity on a machine is reduced. An example of a daemon thread that is continuously running is the garbage collector thread. The JVM exits whenever all **non-daemon** threads have completed, which means that all daemon threads are automatically stopped. To make a thread as a daemon thread in Java → `myThread.setDaemon(true);`

Q 48: 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** **FAQ**

A 48: 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.

Java – Performance and Memory issues

Q. Give me an instance where you made a significant contribution in improving performance ?

There is a good chance that the position you are being interviewed for require someone with skills to identify performance and/or memory issues and ability to optimize performance and solve memory issues. 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. You will find more questions and answers relating to these key areas (i.e. performance and memory issues) in the **Enterprise Java** section and "**How would you go about…"** sections. You could also demonstrate your skills in these key areas by reflecting back on your past experiences as discussed in **Q82** in Java section. Even though **Q82** is a **situational** or **behavioral** question, you can streamline your answer to demonstrate your technical strengths relating to these key areas as well as your behavioral ability to cope with stress.

Q 72: How would you improve performance of a Java application? **PI** **BP** **FAQ**

A 72:

- **Pool valuable system resources** like threads, database connections, socket connections etc. Emphasize 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. Optimize the pool sizes based on system and application specifications and requirements. Having too many threads in a pool also **can result in performance and scalability problems due to consumption of memory stacks** (i.e. each thread has its own stack. Refer **Q34**, **Q42** in Java section) and **CPU context switching** (i.e. switching between threads as opposed to doing real computation.).

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

  Most applications need to retrieve data from and save/update data into one or more databases. Database calls are remote calls over the network. In general data should be **lazily loaded** (i.e. load only when required as opposed to pre-loading from the database with a view that it can be used later) from a database to conserve memory but there are use cases (i.e. need to make several database calls) where **eagerly loading** data and caching can improve performance by minimizing network trips to the database. Data can be eagerly loaded with a help of SQL scripts with complex joins or stored procedures and cached using third party frameworks or building your own framework. At this point your interviewer could intercept you and ask you some pertinent questions relating to caching like:

  Q: How would you refresh your cache?
  A: You could say that one of the two following strategies can be used:

  1. **Timed cache** strategy where the cache can be replenished periodically (i.e. every 30 minutes, every hour etc). This is a simple strategy applicable when it is acceptable to show dirty data at times and also the data in the database does not change very frequently.

  2. **Dirty check** strategy where <u>your application is the only one which can mutate</u> (i.e. modify) the data in the database. You can set a "isDirty" flag to true when the data is modified in the database through your application and consequently your cache can be refreshed based on the "isDirty" flag.

  Q: How would you refresh your cache if your database is shared by more than one application?
  A: You could use one of the following strategies:

  1. **Database triggers**: You could use database triggers to communicate between applications sharing the same database and write pollers which polls the database periodically to determine when the cache should be refreshed. (Refer **Q102** in Enterprise section)

  2. **XML messaging** (Refer **Enterprise – JMS** subsection in Enterprise section) to communicate between other applications sharing the same database or separate databases to determine when the cache should be refreshed.

- **Optimize your I/O operations:** use buffering (Refer **Q25** 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**).

- **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 **Q73** in Java section) to minimize 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 **Q21** in Java section and use **static factory methods** instead of constructors to recycle immutable objects as discussed in **Q16** 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 minimize the number of objects created in your code:

  - 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 selected cases it can represent a significant performance gain. Use **flyweight** design pattern to create a pool of shared objects. Flyweights are typically instantiated by a flyweight factory that creates a limited number of flyweights based on some criteria. Invoking object does not directly instantiate flyweights. It gets it from the flyweight factory, which checks to see if it has a flyweight that fits a specific criteria (e.g. with or without GST etc) in the pool (e.g. HashMap). If the flyweight exists then return the reference to the flyweight. If it does not exist, then instantiate one for the specific criteria and add it to the pool (e.g. HashMap) and then return it to the invoking object.

  - 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.

  - 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 **Q15** 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 **Q17, Q21** in Java Section).

  - Minimize 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" constructs in frequently accessed methods.

  - Do not compute constants inside a large loop. Compute them outside the loop. For applets compute it in the init() method. Avoid nested loops (i.e. a "for" loop within another "for" loop etc) where applicable and make use of a *Collection* class as discussed in "**How can you code better without nested loops ?**" -- **Q17** in Java section.

  - 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 **Q25** in Java section).

  - Minimize calls to Date, Calendar, etc related classes. **For example:**

```
//Inefficient code
public boolean isInYearCompanyWasEstablished(Date dateSupplied) {
    Calendar cal = Calendar.getInstance();
    cal.set(1998, Calendar.JAN, 01,0,0,0); //Should be read from a .proprerties file
    Date   yearStart = cal.getTime();
    cal.setTime(1998,Calendar.DECEMBER, 31,0,0,0);//Should be read from .properties.
    Date yearEnd = cal.getTime();
    return   dateSupplied.compareTo(yearStart) >=0 &&
             dateSupplied.compareTo(yearEnd) <= 0;
}
```

The above code is inefficient because every time this method is invoked 1 “Calendar” object and two “Date” objects are unnecessarily created. If this method is invoked 50 times in your application then 50 “Calendar” objects and 100 “Date” objects are created. A more efficient code can be written as shown below using a static initializer block:

```
//efficient code
private static final YEAR START;
private static final YEAR END;

static{
    Calendar cal = Calendar.getInstance();
    cal.set(1998, Calendar.JAN, 01,0,0,0); //Should be read from a .proprerties file
    Date  YEAR START = cal.getTime();
    cal.setTime(1998,Calendar.DECEMBER, 31,0,0,0);//Should be read from .properties.
    Date YEAR_END = cal.getTime();

}

public boolean isInYearCompanyWasEstablished(Date dateSupplied) {
    return   dateSupplied.compareTo(YEAR_START) >=0 &&
             dateSupplied.compareTo(YEAR_END) <= 0;

}
```

No matter, how many times you invoke the method `isInYearCompanyWasEstablished(…)`, only 1 “Calendar” object 2 “Date” objects are created, since the static initializer block is executed only once when the class is loaded into the JVM.

- Minimize JNI calls in your code.

Q. When in the development process should you consider performance issues?

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

Q. When designing your new code, what level of importance would you give to the following attributes?

-- Performance
-- Maintainability
-- Extendibility
-- Ease of use
-- Scalability

You should not compromise on architectural principles for just performance. You should make effort to write architecturally sound programs as opposed to writing only fast programs. If your architecture is sound enough then it would allow your program not only to scale better but also allows it to be optimized for performance if it is not fast enough. If you write applications with poor architecture but performs well for the current requirements, what will happen if the requirements grow and your architecture is not flexible enough to extend and creates a maintenance nightmare where fixing a code in one area would break your code in another area. This will cause your application to be re-written. So you should think about extendibility (i.e. ability to evolve with additional requirements), maintainability, ease of use, performance and scalability (i.e. ability to run in multiple servers or machines) during the design phase. List all possible design alternatives and pick the one which is conducive to sound design architecturally (i.e. scalable, easy to use, maintain and extend) and will allow it to be optimized later if not fast enough. You can build a vertical slice first to validate the above mentioned design attributes as discussed in **Q82** in the Java section.

Q. Rank the above attributes in order of importance?

There is no one correct answer for this question. **[Hint]** It can vary from application to application but typically if you write 1 - extendable, 2 - maintainable and 3 – ease of use code with some high level performance considerations, then it should allow you to optimize/tune for 4 - performance and 5 - scale. But if you write a code, which only performs fast but not flexible enough to grow with the additional requirements, then you may end up re-writing or carrying out a major revamp to your code. Refer **SOA** (Service Oriented Architecture) **Q15** in How would you go about… section.

Q 73: How would you detect and minimize memory leaks in Java? **MI** **BP** **FAQ**

A 73: 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).

Minimizing 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 **Q38** 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 **Q37** 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 74: Why does the JVM crash with a core dump or a Dr.Watson error? **MI**

A 74: 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 J2SE 5.0 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:

Java – Personal and Behavioral/Situational

Q 75: Did you have to use any design patterns in your Java project? **DP** **FAQ**

A 75: Yes. Refer **Q12 [Strategy], Q16 [Iterator], Q24 [Decorator], Q36 [Visitor], Q51 [Singleton], Q52 [Factory], Q58 [Command], Q61 [Composite],** and **Q63 [MVC-Model View Controller]** in Java section and **Q11, Q12** in How would you go about… section for a detailed discussion on design patterns with class diagrams and examples.

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 interviews. 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. You should not just apply a pattern for the sake of learning or applying a pattern because it could become an anti-pattern.

IMPORTANT: Technical skills alone are not sufficient for you to perform well in your interviews and progress in your career. Your technical skills must be complemented with business skills (i.e. knowledge/understanding of the business, ability to communicate and interact effectively with the business users/customers, ability to look at things from the user's perspective as opposed to only technology perspective, ability to persuade/convince business with alternative solutions, which can provide a win/win solution from users' perspective as well as technology perspective), ability to communicate effectively with your fellow developers, immediate and senior management, ability to work in a team as well as independently, problem solving/analytical skills, organizational skills, ability to cope with difficult situations like stress due to work load, deadlines etc and manage or deal with difficult people, being a good listener with the right attitude (It is sometimes possible to have "**I know it all attitude**", when you have strong technical skills. This can adversely affect your ability to be a good listener, ability to look at things in a different perspective, ability to work well in a team and consequently your progression in your career) etc. Some of these aspects are covered below and should be prepared for prior to your job interview(s).

Q 76: 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? **FAQ**

A 76: **[Hint:]** Pick your recent projects and **enthusiastically** brief on it. Interviewer will be looking for how passionate you are about your past experience and achievements. Also is imperative that during your briefing, you demonstrate on a high level(without getting too technical) how you applied your skills and knowledge in some of the following key areas:

- Design concepts and design patterns: **How you understood and applied them**.
- Performance and memory issues: **How you identified and fixed them**.
- Exception handling and best practices: **How you understood 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 **Q7, Q8, Q9, Q10, Q11** etc
- **Design Patterns**: Refer **Q12, Q16, Q24, Q36, Q51, Q52, Q58, Q61**, and **Q63** in Java section and **Q11, Q12** in "How would you go about…?" section for a more detailed discussion.
- **Performance issues**: Refer **Q25, Q72** etc
- **Memory issues**: Refer **Q37, Q38, Q42, Q73,** and **Q74**
- **Exception Handling**: Refer **Q39, Q40** etc
- **Multi-threading (Concurrency issues)**: Refer **Q15, Q17, Q21, Q34, Q42** and **Q46** 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**.**

The job seekers should also ask questions to make an impression on the interviewer. Write out specific questions you want to ask and then look for opportunities to ask them during the interview. For example:

- Do you have any performance or design related issues? → Succinctly demonstrate how you would go about solving them or how you solved similar problems in your previous assignments.

- Do you follow any software development processes like agile methodology, XP, RUP etc? → Briefly demonstrate your experience, understanding and/or familiarity with the development methodology of relevance.

- Do you use any open source frameworks like Spring, Hibernate, Tapestry etc? Any build tools like Ant, Maven etc, and testing tools like JUnit etc → briefly demonstrate your experience, understanding and/or familiarity with the framework(s) of relevance.

Many interviewers end with a request to the applicant as to whether they have anything they wish to add. This is an opportunity for you to end on a positive note by making succinct statements about why you are the best person for the job by demonstrating your understanding of the key areas and how you applied them in your previous jobs.

Reflect back on your past jobs and pick two to five instances where you used your skills in the key areas very successfully.

Q 77: Why are you leaving your current position? **FAQ**
A 77: [Hint]

- Do not criticize your previous employer or co-workers 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 78: What do you like and/or dislike most about your current and/or last position? **FAQ**
A 78: [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 co-workers etc.

It is **safe to say**:

- You like challenges.
- Opportunity to grow into design, architecture, performance tuning etc
- Opportunity to learn and/or mentor junior developers..
- 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 79: How do you handle pressure? Do you like or dislike these situations? **FAQ**
A 79: **[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 80: 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? **FAQ**
A 80: [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 practice 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 81: What are your career goals? Where do you see yourself in 5-10 years? **FAQ**
A 81: **[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.

Situational questions: The open-ended questions like last two questions are asked by interviewers to identify specific characteristics like taking initiative, performance standards, accountability, adaptability, flexibility, sensitivity, communication skills, ability to cope stress etc. These questions are known as behavioral or situational questions. This

behavioral technique is used to evaluate a candidate's future success from past behaviors. The answers to these questions must describe in detail a particular situation like an event, a project or an experience and how you acted on that situation and what the results were. Prepare your answers prior to the interview using the "**S**ituation **A**ction **R**esult (SAR)" approach and avoid fabricating or memorizing your answers. You should try to relate back to your past experiences at your previous employments, community events, sporting events etc. Sample questions and answers are shown below:

Q 82: Give me an example of a time when you set a goal and were able to achieve it? Give me an example of a time you showed initiatiative and took the lead? Tell me about a difficult decision you made in the last year? Give me an example of a time you motivated others? Tell me about a most complex project you were involved in? **FAQ**

A 82:

Situation: When you were working for the ZCC Software Technology Corporation, the overnight batch process called the "Data Pacakager" was developed for a large fast food chain which has over 100 stores. This overnight batch process is responsible for performing a very database intensive search and compute changes like cost of ingredients, selling price, new menu item etc made in various retail stores and package those changes into XML files and send those XML data to the respective stores where they get uploaded into their point of sale registers to reflect the changes. This batch process had been used for the past two years, but since then the number of stores had increased and so did the size of the data in the database. The batch process, which used to take 6-8 hours to complete, had increased to 14-16 hours, which obviously started to adversely affect the daily operations of these stores. The management assigned you with the task of improving the performance of the batch process to 5-6 hours (i.e. suppose to be an overnight process).

Action: After having **analyzed the existing design** and code for the "Data Packager", **you had to take the difficult decision** to let the management know that this batch process needed to be re-designed and re-written as opposed to modifying the existing code, since it was poorly designed. It is hard to extend, maintain (i.e. making a change in one place can break the code some where else and so on) and had no object reuse through caching (makes too many unnecessary network trips to the database) etc. The management was not too impressed with this approach and concerned about the time required to rewrite this batch process since the management had promised the retail stores to provide a solution within 8-12 weeks. **You took the initiative and used your persuasive skills** to **convince the management** that you would be able to provide a re-designed and re-written solution within the 8-12 weeks with the assistance of 2-3 additional developers and two testers. You were entrusted with the task to rewrite the batch process and **you set your goal to complete the task in 8 weeks**. You decided to build the software iteratively by building individual vertical slices as opposed to the big bang waterfall approach [Refer subsection "**Enterprise – Software development process**" in Enterprise – Java section]. You redesigned and wrote the code for a typical use case from end to end (i.e. full vertical slice) within 2 weeks and subsequently carried out functional and integration testing to iron out any unforeseen errors or issues. Once the first iteration is stable, **you effectively communicated** the architecture to the management and to your fellow developers. **Motivated** and **mentored** your fellow developers to build the other iterations, based on the first iteration. At the end of iteration, it was tested by the testers, while the developers moved on to the next iteration.

Results: After having **enthusiastically** worked to your plan with **hard work**, **dedication** and **teamwork**, you were able to have the 90% of the functionality completed in 9 weeks and spent the next 3 weeks fixing bugs, tuning performance and coding rest of the functionality. The fully functional data packager was completed in 12 weeks and took only 3-4 hours to package XML data for all the stores. The team was under pressure at times but you made them believe that it is more of a **challenge as opposed to think of it as a stressful situation**. The newly designed data packager was also easier to maintain and extend. The management was impressed with the outcome and rewarded the team with an outstanding achievement award. The performance of the newly developed data packager was further improved by 20% by tuning the database (i.e. partitioning the tables, indexing etc).

Q 83: Describe a time when you were faced with a stressful situation that demonstrated your coping skills? Give me an example of a time when you used your fact finding skills to solve a problem? Describe a time when you applied your analytical and/or problem solving skills? **FAQ**

A 83:

Situation: When you were working for the Surething insurance corporation pty ltd, you were responsible for the migration of an online insurance application (i.e. external website) to a newer version of application server (i.e. the current version is no longer supported by the vendor). The migration happened smoothly and after a couple of days of going live, you started to experience "OutOfMemoryError", which forced you to restart the application server every day. This raised a red alert and the immediate and the senior management were very concerned and consequently constantly calling for meetings and updates on the progress of identifying the root cause of this issue. This has created a stressful situation.

In short, arrive on time, be polite, firm hand with a smile and do not act superior, act interested and enthusiastic but not desperate, make eye contact at all times, ask questions but should not over do it by talking too much, it is okay to be nervous but try not to show it and be honest with your answers because you are not expected to know the answers for all the technical questions. Unless asked, do not talk about money and find every opportunity to sell your technical, business and interpersonal skills without over doing it. Finish the interview with a positive note by asking about the next steps if not already mentioned, a firm hand shake with a "thank you for the interviewer's time" with an eye contact and a smile.

General Tip #1:

- 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], business skills, and interpersonal skills 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. [**Technical skills must be complemented with good interpersonal skills**]

General Tip #2:

Prepare a skills/knowledge matrix in your Resume. This is very useful for someone who gained wide range of skills/knowledge in a short span by being a pro-active learner (e.g. extra reading, additional projects, outside work development projects etc). **For example:**

Java 1.3 – 5.0	**18 months**
Servlets / JSP	**12 months**
J2EE (EJB, JMS, JNDI etc)	**12 months**
XML, XSD, XSLT etc	**6 months**
Hibernate	**6 months**
OOA & OOD	**12 months**
UML	**4 months**
Design patterns	**5 months**
SQL	**12 months**

General Tip #3:

Unless you are applying for a position as a junior or a beginner developer, having your resume start with all the Java training and certifications may lead to a misunderstanding that you are a beginner. Your first page should concentrate on your achievements and skills summary (As in **General Tip #2**) to show that you are a skilled professional. **For example:**

- Re-designed the data packager application for the XYZ Corporation, to make it more scalable, maintainable and extendable. [**Shows that you have design skills**]
- Identified and fixed memory leak issues for the master lock application and consequently improved performance by 20% and further improved performance by introducing multi-threading and other performance tuning strategies. Identified and fixed some transactional issues for the Endeavor project, which is a web based e-commerce application. [**Shows that you are a pro-active developer with good understanding of multi-threading, transactional, performance and memory issues. Also shows that you have worked on transactional and multi-threaded systems and have an eye for identifying potential failures.**]
- Received an outstanding achievement award for my design and development work using Java/J2EE at the ABC Corporation. Published an article entitled "Java Tips and Tricks". [**Shows that you take pride in your achievements**]
- Mentored junior developers at JKL Corporation. [**Shows that you are an experienced developer who would like to mentor junior developers and you are not only a technology oriented person but also a people oriented person**].

Reference your achievements and accomplishments with specific examples and/or relevant paperwork (but avoid overloading the hiring manager with paperwork).

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 packages help resolve naming conflicts when different packages have classes with the same names. This also helps you organize files within your project.
- 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). Favor object composition over inheritance.
- When using **implementation inheritance**, make sure that the **subclasses depend only on the behavior of the superclass**, not the actual implementation. An **abstract** base class usually provides an implementation inheritance.
- Favor **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 collections 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.
- The equals() - returns the results of running the equals() method of a user supplied class, which compares the attribute values. The equals() method provides “**deep comparison**” by checking if two objects are logically equal as opposed to the shallow comparison provided by the operator ==.
- The non-final methods **equals**(), **hashCode**(), **toString**(), **clone**(), and **finalize**() are defined in the Object class and are primarily meant for extension. The equals() and hashCode() methods prove to be very important when objects implementing these two methods are added to collections.
- If a class overrides the equals() method, it must implement the hashCode() method as well. If two objects are equal as per the equals() method, then calling the hashCode() method in each of the two objects must return the same hashCode integer result but the reverse is not true (i.e. If two objects have the same hashCode does not mean that they are equal). If a field is not used in equals()method, then it must not be used in hashCode() method.
- When providing a user defined key class for storing objects in *HashMap*, you should override **equals()**, and **hashCode()** methods from the *Object* class.
- Always override the toString() method, but you should override the clone() method very judiciously. The finalize() method should only be used in rare instances as a safety net or to terminate non-critical native resources.
- *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 (i.e. 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, minimizing 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.

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

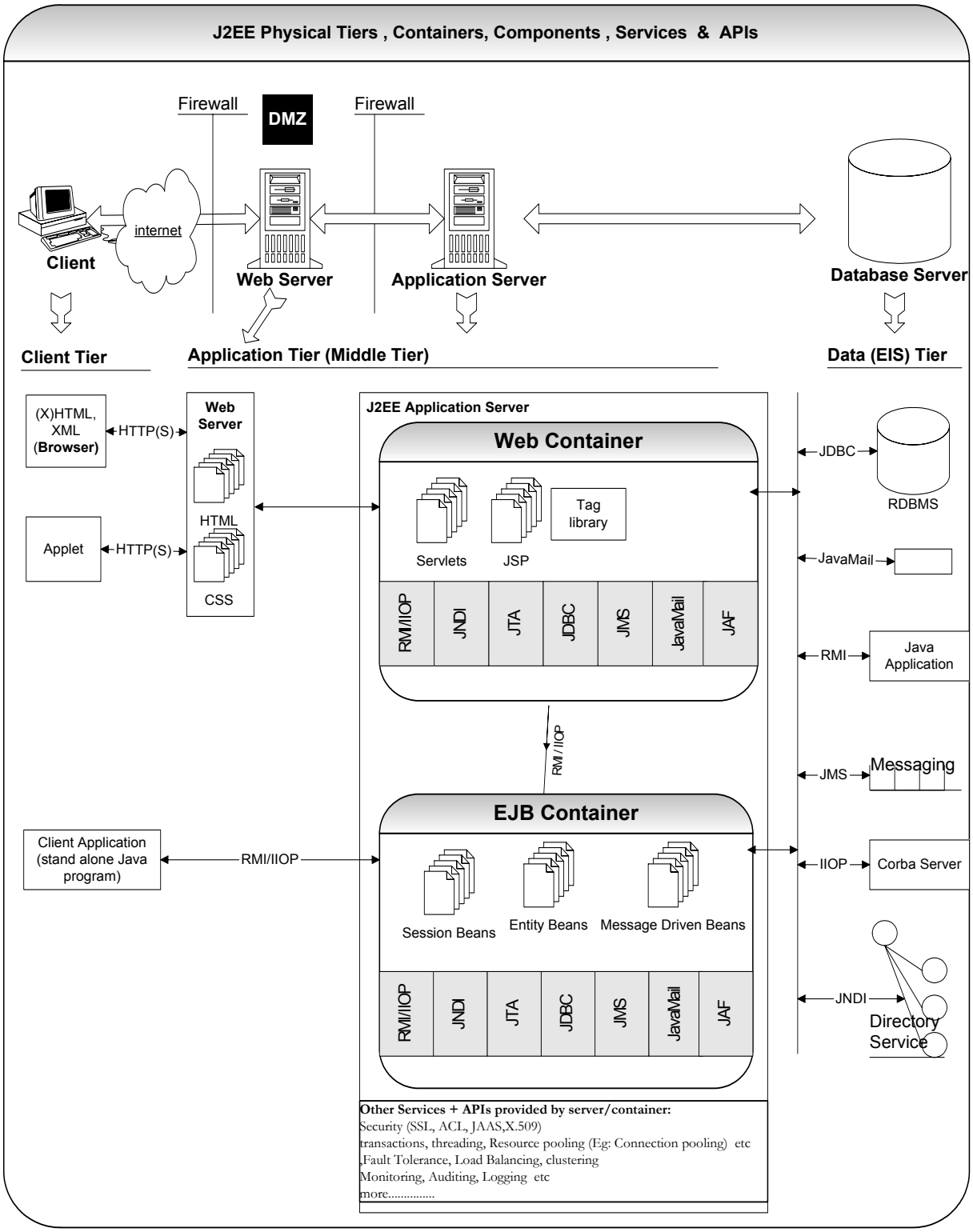
FAQ - Frequently **A**sked **Q**uestions

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

Enterprise - J2EE Overview

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

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)

Q. So what is the difference between a component and a service?

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). A service is a step up from "distributed objects". A service is a function that has a clearly defined service contract (e.g. interface, XML contract) to their consumers or clients, self contained and does not depend on the context or state of other services.

Q. What is a Service Oriented Architecture (SOA)?

SOA is an evolution of the fundamentals governing a component based development. Component based development provides an opportunity for greater code reuse than what is possible with **O**bject **O**riented (**OO**) development. SOA provides even greater code reuse by utilizing OO development, component based development and also by identifying and organizing right services into a hierarchy of composite services. SOA results in loosely coupled application components, in which code is not necessarily tied to a particular database. SOAs are very popular and there is a huge demand exists for development and implementation of SOAs. Refer **Q14** in **How would you go about…?** section for a more detailed discussion on **SOA** and **Web services**.

Q. What are Web and EJB containers?

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.

Q. Why do you need a J2EE server? What services does a J2EE server provide?

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.

Server supports various protocols. **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 summarized as follows:

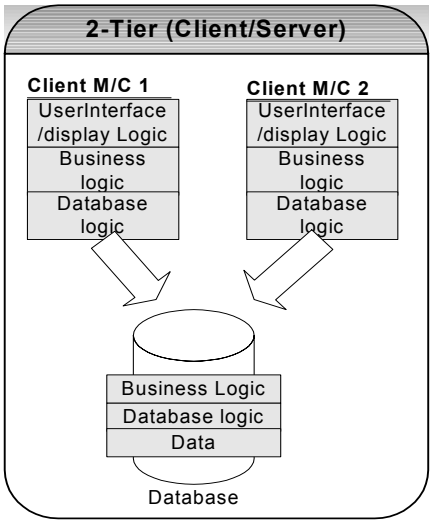
J2EE technology category	API (Application Programming Interface)
Component model technology	**Java Servlet**, JavaServer Pages(**JSP**), Enterprise JavaBeans(**EJB**).
Web Services technology	**JAXP** (Java API for XML Processing), **JAXR** (Java API for XML Registries), **SAAJ** (SOAP with attachment API for Java), **JAX-RPC** (Java API for XML-based RPC), **JAX-WS** (Java API for XML-based Web Services).

Other	**JDBC** (**J**ava **D**ata**B**ase **C**onnectivity), **JNDI** (**J**ava **N**aming and **D**irectory **I**nterface), **JMS** (**J**ava **M**essaging **S**ervice), **JCA** (**J**2EE **C**onnector **A**rchitecture), **JTA** (**J**ava **T**ransaction **A**PI), **JavaMail**, **JAF** (**J**avaBeans **A**ctivation **F**ramework – used by JavaMail), **JAAS** (**J**ava **A**uthentication and **A**uthorization **S**ervice), **JMX** (**J**ava **M**anagement e**X**tensions).

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

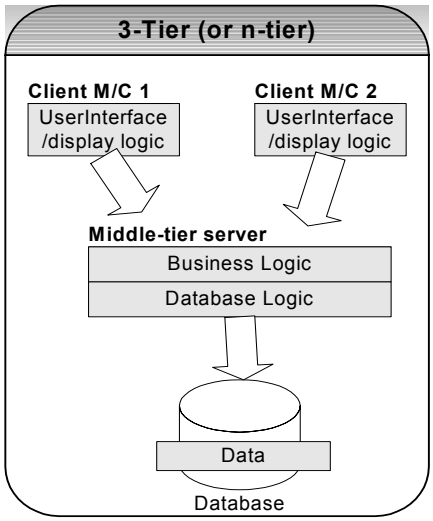
A 02: This is a very commonly asked question. Be prepared to draw some diagrams on the board. The J2EE platform is a multi-tiered system. A tier is a logical or functional partitioning of a system.

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.

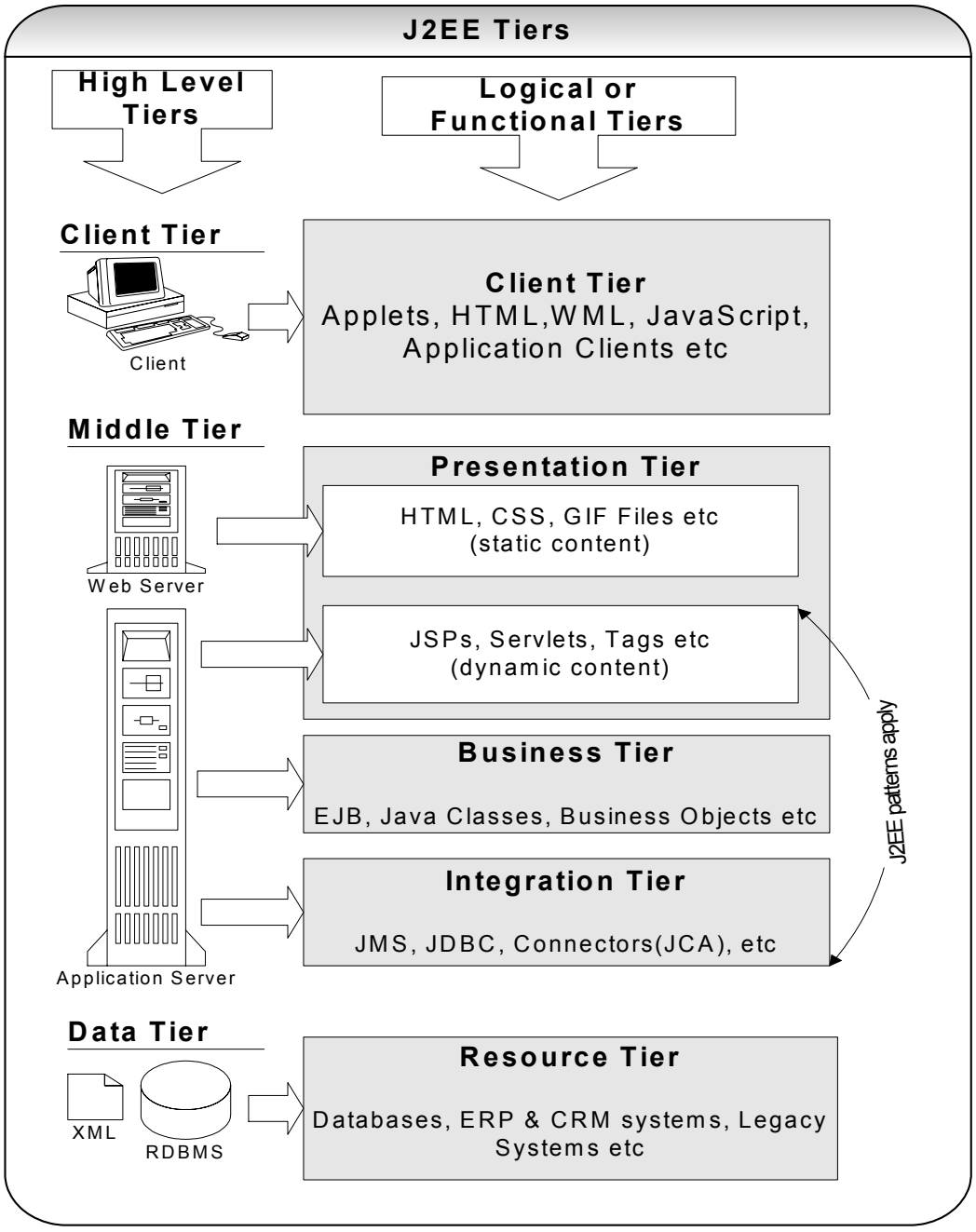
Client tier represents Web browser, a Java or other application, Applet, WAP phone etc. The client tier makes requests to the Web server who will be serving the request by either returning static content if it is present in the Web server or forwards the request to either Servlet or JSP in the application 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 centralized 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 centralized business tier is that same business logic can support different types of clients like browser, WAP (Wireless Application Protocol) client, other stand-alone applications written in Java, C++, C# 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.

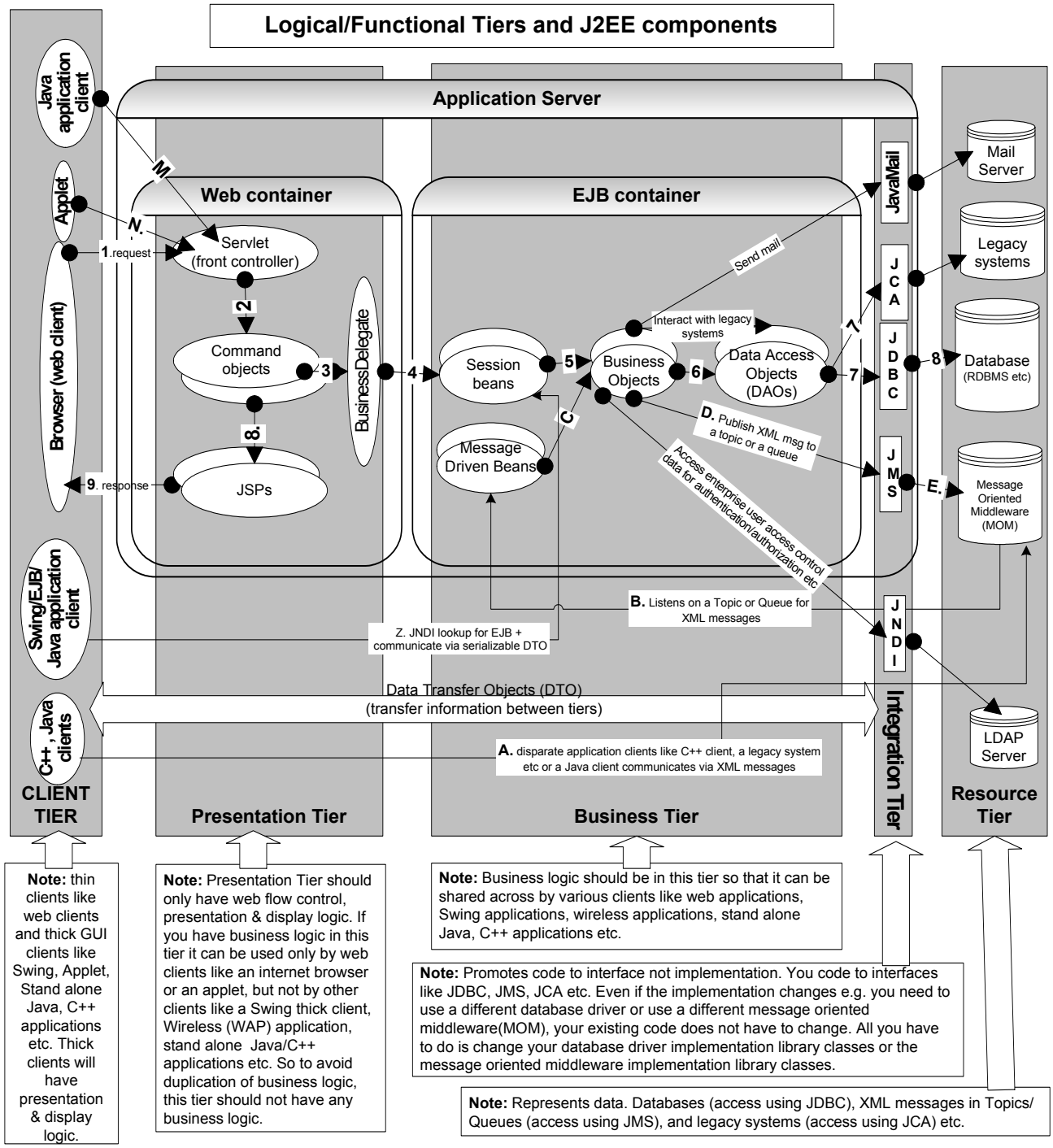


Note: On a high level J2EE can be construed as a **3-tier** system consisting of **Client Tier, Middle Tier** (or Application Tier) and **Data Tier**. But logically or functionally J2EE is a multi-tier (or n-tier) platform.

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:

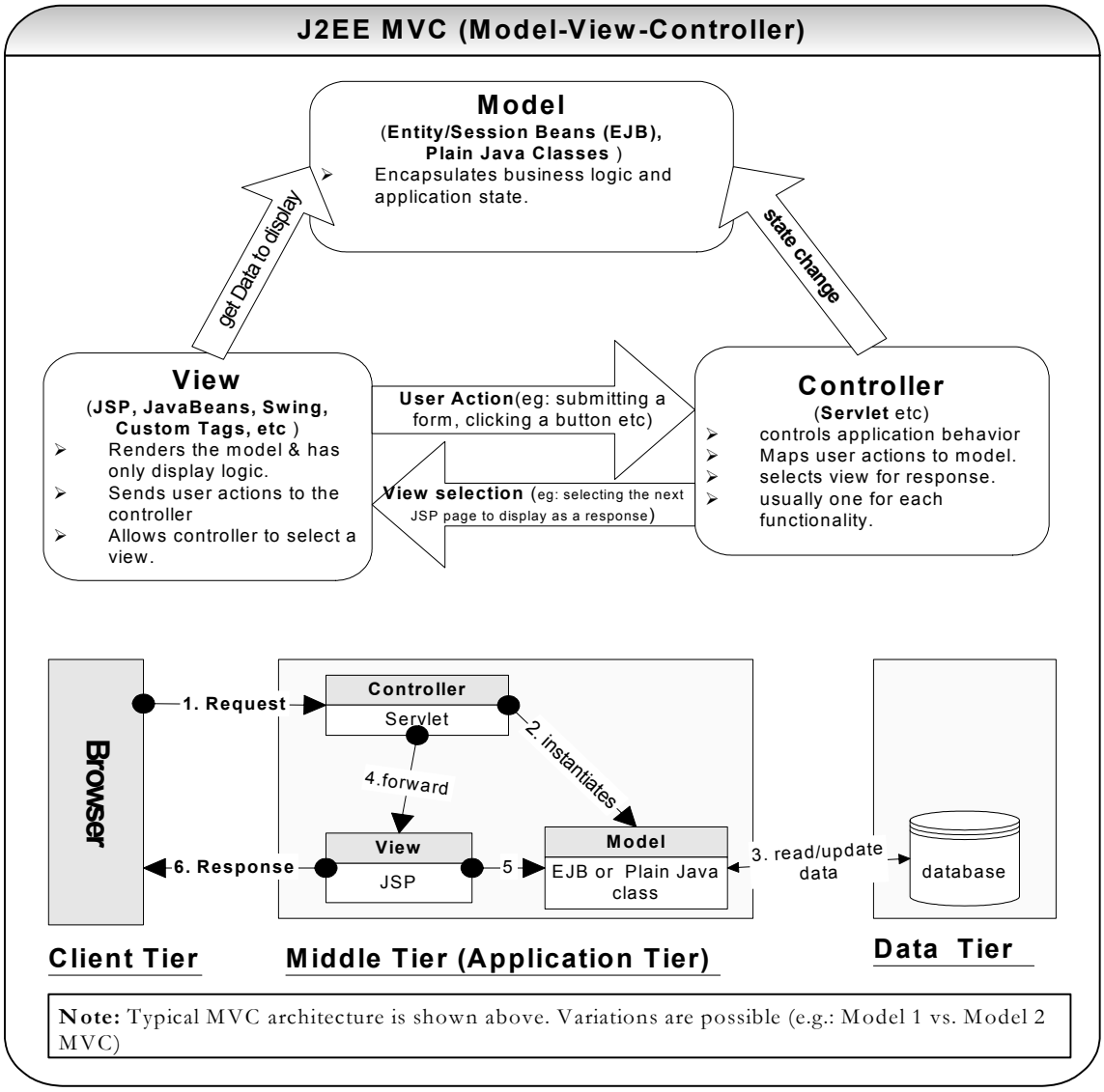
- **Manageability:** Each tier can be monitored, tuned and upgraded independently and different people can have clearly defined responsibilities.
- **Scalability:** More hardware can be added and allows clustering (i.e. horizontal scaling).
- **Maintainability:** Changes and upgrades can be performed without affecting other components.
- **Availability:** Clustering and load balancing can provide availability.
- **Extensibility:** Additional features can be easily added.

The following diagram gives you a bigger picture of the logical tiers and the components.



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

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 minimize the degree of coupling (i.e. promotes loose coupling) between components. It is often used by applications that need the ability to maintain multiple views like HTML, WML, Swing, XML based Web service etc of the same data. Multiple views and controllers can interface with the same model. Even new types of views and controllers can interface with a model without forcing a change in the model design.



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 translates interactions with the view into actions to be performed by the model. User interactions in a Web application appear as GET and POST HTTP requests. The actions performed by a model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

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 minimizes 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.

If you are applying for a senior developer or an architect level role, you should at least know the more common design patterns like:

-- Factory - **Q52** in Java section, **Q11** in How would you go about… section.
-- Singleton - **Q51** in Java section, **Q11** in How would you go about… section.
-- Proxy - **Q52, Q62** in Enterprise Java section, **Q11** in How would you go about… section.
-- Command - **Q58** in Java section, **Q27**, **Q110**, **Q116** in Enterprise Java section, **Q11** in How would you go about… section.
-- Template method - **Q110**, **Q116** in Enterprise Java section, **Q11** in How would you go about… section.
-- Decorator - **Q24** in Java section, **Q11** in How would you go about… section.
-- Strategy - **Q64** in Java section, **Q11** in How would you go about… section.
-- Adapter - **Q110**, **Q116** in Enterprise Java section, **Q11** in How would you go about… section.
-- Façade - **Q84** in Enterprise Java section, **Q11**, **Q12**, **Q15** (i.e. in **SOA**) in How would you go about… section.
-- Business delegate – **Q83** in Enterprise Java section.
-- MVC - **Q63** in Java section, **Q3**, **Q27** in Enterprise Java sections.
-- DAO - **Q41** in Enterprise Java section.

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.

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

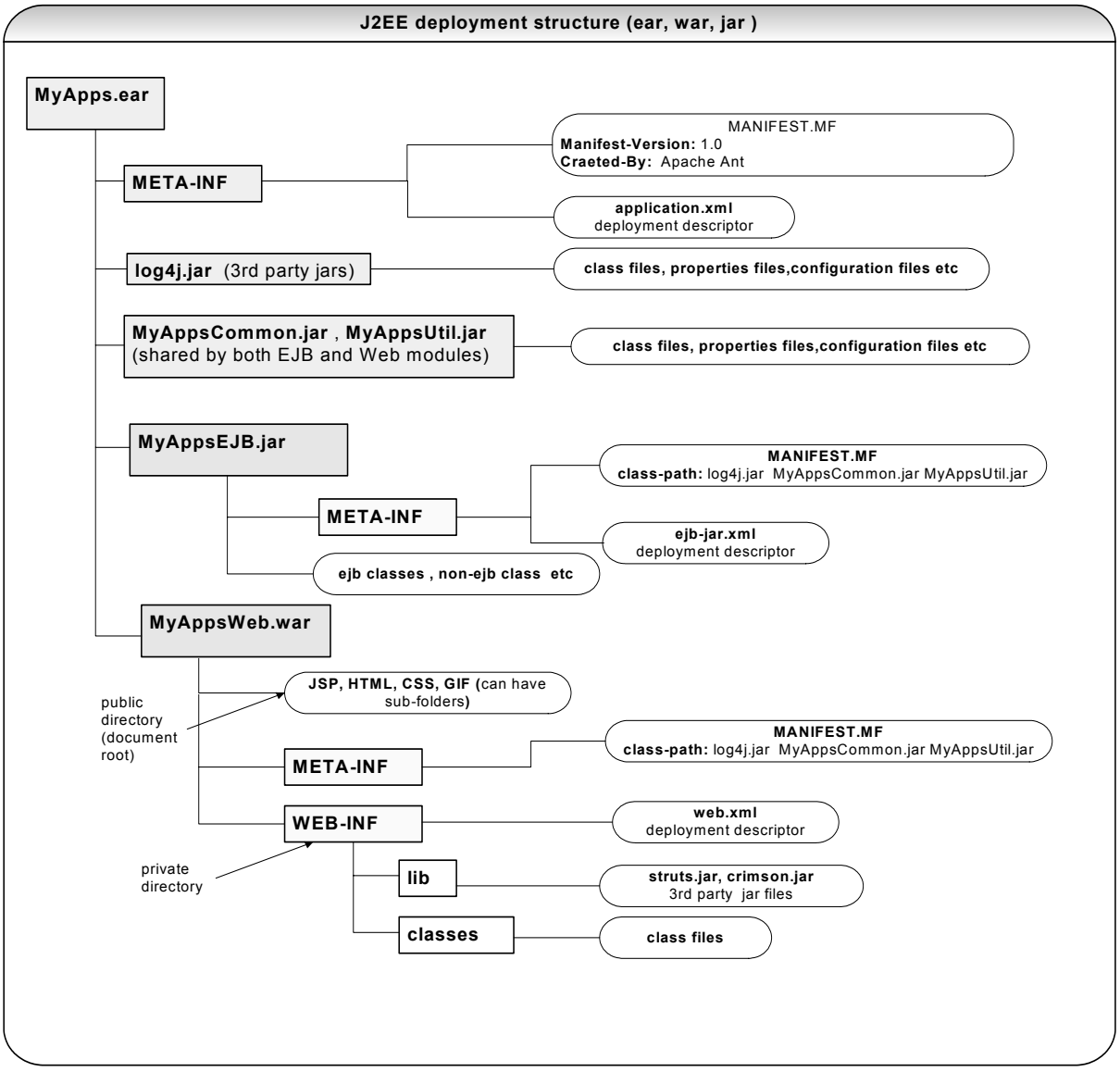
A 07: The ear, war and jar are standard application deployment archive files. Since they are a standard, any application server (at least in theory) will know how to unpack and deploy them.

An EAR file is a standard JAR file with an ".ear" extension, named from **E**nterprise **AR**chive 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, Servlets and static content like HTML, CSS, GIF etc). The J2EE application client's class files are also stored in a JAR file. EARs, JARs, and WARs all contain one or more XML-based deployment descriptor(s).

Deployment Descriptors

A deployment descriptor is an XML based text file with an ".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, MyAppsUtil.jar etc are packaged. The application.xml descriptor will distinguish between these two types of 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>manager</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>
```

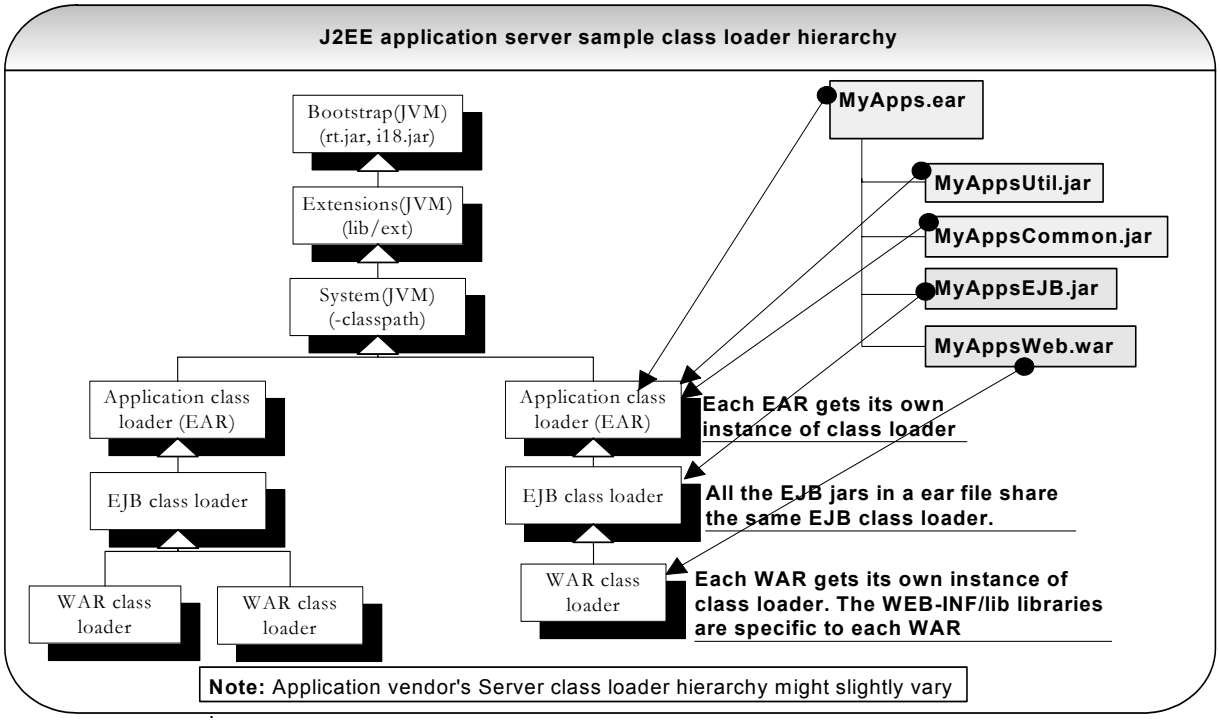
```
  <security-role>
      <description>Advisor</description>
      <role-name>advisor</role-name>
  </security-role>

</web-app>
```

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

A 08: J2EE application server sample class loader hierarchy is shown below. (Also refer to **Q5** 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 in the hierarchy 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 also 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 (e.g. MyAppsUtility.jar) and common (e.g. MyAppsCommon.jar) jars etc. Any application specific exception like *MyApplicationException* thrown by an EJB module should be caught by a Web module. So the exception class *MyApplicationException* is shared by both Web and EJB modules.

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 (i.e. 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 **Q5** 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 behavior 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**).

Tech Tip #4:

Q. What do the terms internationalization(i18n) and localization(l10n) mean, and how are they related? Localization (aka **l10n**, where 10 is the number of letters between the letter 'l' and the letter 'n' in the word localization) refers to the adaptation of an application or a component to meet the language, cultural and other requirements to a specific locale (i.e. a target market). Internationalization (aka i18n, where 18 is the number of letters between the letter 'i' and the letter 'n' in the word internationalization) refers to the process of designing a software so that it can be localized to various languages and regions cost-effectively and easily without any engineering changes to the software. A useful website on i18n is http://www.i18nfaq.com.

Q. What are the characteristics of an internalized program?

-- The same executable can run worldwide without having to recompile for other or new languages.
-- Text messages and GUI component labels are not hard-coded in the program. Instead they are stored outside the source code in ".properties" files and retrieved dynamically based on the locale.
-- Culturally dependent data such as dates and currencies, appear in formats that conform to end user's region and language. (e.g. USA → mm/dd/yyyy, AUS → dd/mm/yyyy).

Q. What are the different types of data that vary with region or language?

Messages, dates, currencies, numbers, measurements, phone numbers, postal addresses, tax calculations, graphics, icons, GUI labels, sounds, colors, online help etc.

Q. What is a Locale? A Locale has the form of xx_YY (**xx** – is a two character language code && **YY** is a two character country code. E.g. en_US (English – United States), en_GB (English - Great Britain), fr_FR (french - France). The *java.util.Locale* class can be used as follows:

```
Locale locale1 = new Locale("en", "US");
Locale locale2 = Locale.US;
Locale locale3 = new Locale("en");
Locale locale4 = new Locale("en", "US", "optional"); // to allow the possibility of more than one
                                                     // locale per language/country combination.

locale2.getDefault().toString();         //  en US
locale2.getLanguage();                   //  "en"
locale2.getCountry();                    //  "US"
```

Resource bundles can be created using the locale to externalize the locale-specific messages:

Message_en_US.properties

```
   Greetings = Hello
```

Message_fr_FR.properties

```
   Greetings = Bonjour
```

These resource bundles reside in classpath and gets read at runtime based on the locale.

```
Locale currentLoc = new Locale("fr", "FR");
ResourceBundle messages = ResourceBundle.getBundle("Message", currentLoc);
System.out.println(messages.getString("Greetings"));  //prints Bonjour
```

Note: When paired with a locale, the closest matching file will be selected. If no match is found then the default file will be the Message.properties. In J2EE, locale is stored in HTTP session and resource bundles (stored as *.properties files under WEB-INF/classes directory) are loaded from the web.xml deployment descriptor file. Locale specific messages can be accessed via tags (e.g. Struts, JSTL etc).

The ***java.text*** package consists of classes and interfaces that are useful for writing internationalized programs. By default they use the default locale, but this can be overridden. E.g. *NumbeFormat*, *DateFormat*, *DecimalFormat*, *SimpleDateFormat*, *MessageFormat*, *ChoiceFormat*, *Collator* (compare strings according to the customary sorting order for a locale) etc.

DateFormat:

```
Date now = new Date();
Locale locale = Locale.US;

String s = DateFormat.getDateInstance(DateFormat.SHORT, locale).format(now);
```

NumberFormat:

```
NumberFormat usFormat = NumberFormat.getInstance(Locale.US);
String s1 = usFormat.format(1785.85);  // s1 → 1,785.85

NumberFormat germanyFormat = NumberFormat.getInstance(Locale.GERMANY);
String s2 = germanyFormat.format(1785.85);  // s2 → 1.785,85
```

To use default locale:

```
NumberFormat.getInstance();
NumberFormat.getPercentInstance();
NumberFormat.getCurrencyInstance();
```

To use specific locale:

```
NumberFormat.getInstance(Locale.US);
NumberFormat.getCurrencyInstance(myLocale);
```

INDEX

To purchase the complete book: http://www.lulu.com/content/192463
(2nd edition)

Home page: http://www.lulu.com/java-success