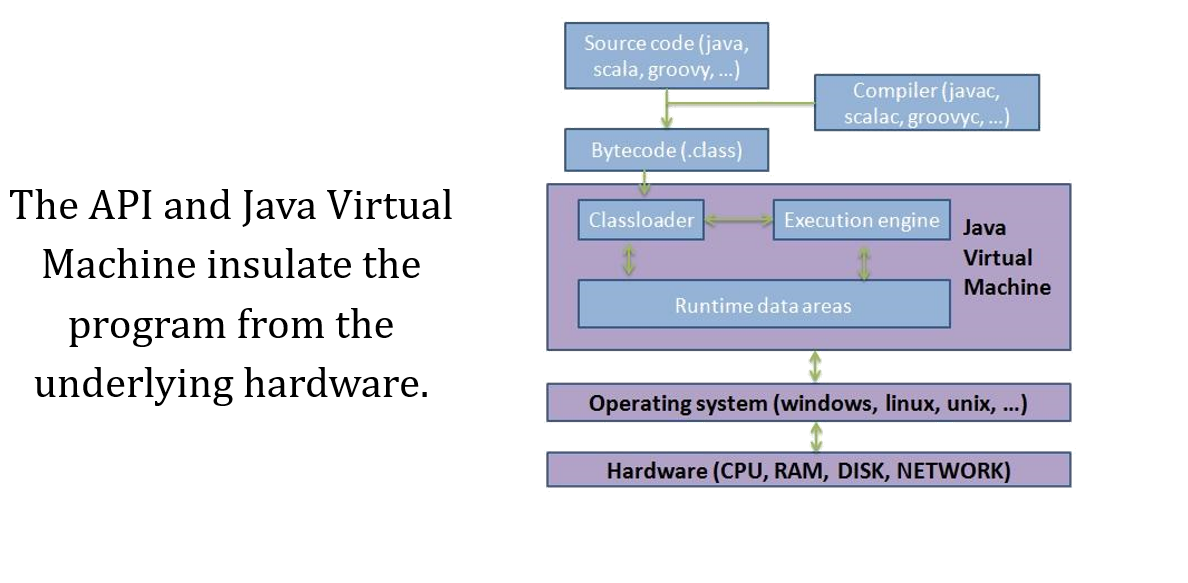
**Java SE**

1. **What is Java?**

Java technology is both a programming language and a platform from ***Oracle Corporation****.*

1. **Why is Java called a “Platform”?**

A platform is basically the hardware or software environment in which a program runs. Java provides software-based platform i.e., JVM, which can run applications developed using the Java Programming Language.



1. **What are the features of Java?**

Features of Java

**1) Simple**

Simple syntax. No pointers, no multiple inheritance with the classes which causes ambiguity error. For almost every task API (Application Programming Interface) is available; Programmers just need to know how to use that API.

**2) Object Oriented**

Java is strongly object oriented as it does not allow features like global data, friend functions which are against OOP principles.

**3) Automatic Garbage Collection:**

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a reference to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

**4) Robust**

Robust means safe. Java puts a lot of emphasis on early checking for possible errors, as Java compilers are able to detect many problems that would first show up during execution time in other languages.

It provides a powerful exception handling and type checking mechanism as compared to other programming languages.

**5) Platform Independent**

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be a compile-once, run-anywhere language. On compilation Java programs are compiled into bytecode. This bytecode is platform independent and can be run on any machine. Any machine with Java Runtime Environment can run Java Programs.

**6) Secure**

If a bytecode contains any virus or malicious code, JVM will not execute it. This feature saves your system especially when you download java code and try to execute.

**7) Multi-Threading**

Java multithreading feature makes it possible to write programs that can do many tasks simultaneously.

**8) Portable**

Java Byte code can be carried to any platform.

**9) Architectural Neutral**

No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types is same on all the platforms.

**10) High Performance**

Java enables high performance with the use of Just-In-Time (JIT) compiler.

1. **How is java platform independent?**

Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS.

1. **Which version of java have you learned? Name some of the new features added to it**

Java 8 version. Some of the new features are:

1. default methods and static methods inside interface
2. lambda expressions
3. method reference
4. stream api
5. **What is native code?**

Native code is computer programming (code) that is compiled to run with a particular processor (such as an Intel x86-class processor) and its set of instructions.

1. **What is the difference between JDK and JRE?**

**JRE** is an acronym for Java Runtime Environment. It is used to provide a runtime environment. It contains JVM + API ( jars and dlls)

**JDK** is an acronym for Java Development Kit. It contains JRE + development tools like compiler, debugger etc.

1. **What is JVM? What does it do?**

A Java virtual machine (JVM), an implementation of the Java Virtual Machine Specification, interprets compiled Java binary code (called bytecode) for a computer's processor (or "hardware platform") so that it can perform a Java program's instructions. Java was designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the programmer for each separate platform.

JVMs are available for various platforms (i.e.JVM is platform dependent)

It is:

* **A specification** where working of Java Virtual Machine is specified. But the implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
* **An implementation:** Its implementation is different for different platforms.
* **Runtime Instance** Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

The JVM performs following operation:

* Loads code
* Verifies code
* Executes code

1. **Why is a JVM called a “virtual machine”?**

The JVM is "virtual" because it is generally implemented in software on top of a "real" hardware platform and operating system. All Java programs are compiled for the JVM. Therefore, the JVM must be implemented on a particular platform before compiled Java programs will run on that platform.

1. **What is the lifetime of a Java Virtual Machine?**

When a Java application starts, a runtime instance of JVM is born. When the application completes, the instance dies.

1. **What are the main components of JVM? Explain them Or Explain JVM Architecture**

a) **Class Loader Subsystem b) Various Runtime Data Areas c) Execution Engine**

**Class loader Subsystem:**

Several parts

* **Loading**
* **Linking [ verify, prepare and resolve]**
* **Initializing**

a) **Loading**

**Bootstrap class loader**

Load classes from “rt.jar” ie. All core java API classes are loaded.

**Extension class loader**

Loads classes from the “ext” folder.

**Application class loader**

Loads classes from application level class path – set inside environment

Amongst all these “Bootstrap class loaders” will get highest priority. If “Bootstrap class loader is unable to load classes then “Extension class loader” will load the classes. If “Extension class loader” is also unable to load class files, then, “application class loader” will try to load the class file.

b) **Linking**

**Verify**

The Bytecode Verifier is going to check whether your generated bytecode is proper or not. I.e. Whether it contains any virus or malicious code and whether class file format is compatible with JVM class specification. If verification fails then we will get “Verify Error”. Java programs are always secure. You can execute on any machine happily because the Bytecode Verifier is going to take care ,is it valid or not.

**Prepare**

For static variables memory will be allocated and assigned with the default values

**Resolve**

It is an eager resolution if any.

c) **Initialization**

Static variables are assigned with the values u have initialized. Static blocks are executed in the textual order. After Initialization only “class loading” completed successfully by “Class Loader Subsystem”. ClassLoader Subsystem is responsible for loading our classes. In order to load the classes, some memory is required.

There are various Runtime DataAreas present inside JVM.

1. **Method area:**- will consists of following things:

* **class bytecode**
* **class loader who loads the class information**
* **reference to class Class**
* **static variables**
* **various literals (String,Integer,float et**c)

**Type Information**

* For each type it loads, a Java virtual machine must store the following kinds of information in the method area:
* The fully qualified name of the type
* The fully qualified name of the type's direct superclass (unless the type is an interface or class java.lang.Object, neither of which have a superclass)
* Whether or not the type is a class or an interface
* The type's modifiers ( some subset of` public, abstract, final)
* An ordered list of the fully qualified names of any direct super-interfaces

**Method Information**

For each method declared in the type, the following information must be stored in the method area. As with fields, the order in which the methods are declared by the class or interface must be recorded as well as the data. Here's the list:

* The method's name
* The method's return type (or void)
* The number and types (in order) of the method's parameters
* The method's modifiers (some subset of public, private, protected, static, final, synchronized, native, abstract)

Since Java8, there is no “Method Area” , Now it is called a “metaspace”.

1. **Heap area**: Objects and corresponding instance variables will be stored in heap area.

**Method area and Heap area are per JVM.**

1. **Stack area**: For every thread a separate runtime stack will be created. Each entry (method name) in the stack is called a “Stack frame”. For every thread, one runtime stack is there. So if inside JVM 10 threads are running, 10 stacks are there. The data which is stored inside “Stack Memory” is always a “Thread-Safe”. It is because that data is available only for that thread not for other thread and vice-versa. In the case of a recursive function if you forget to put a condition it goes into an infinite loop. That is it will go on creating stack frame over stack frame over stack frame. That means u will go out of space in stack area memory and u will get “java.lang.StackOverflowError”
2. **PC register:- Program Counter Register**

Every thread has got a different PC register which stores pointers to the next line to be executed by JVM.

i.e. to hold the address of the next executing instruction.

Suppose u r running three threads T1,T2 and T3.

In this case there will be 3 program counters each for T1, T2 and T3.

1. **Native method stack:-** if threads are invoking any native method, then for every thread a different stack will be created inside native method stack.

PC registers, Stack and Native Method Stack are per thread. So Thread1 can’t see what is there in the stack frame created for Thread2 and vice-versa. That means whatever happens in method execution generally is thread-safe.

At the same time “Method Area or metaspace(in java8) “ and “heap area” are not per thread. These are created per JVM.

**Execution engine:-**

The Execution area will communicate with “Memory areas”. i.e. once a data area is loaded which is an instruction to be executed, i.e. the current instruction to be executed is ready ( using PC register) what happens is that “Java Interpreter” interprets the current instruction that is there in the bytecode and executes it.

Execution Engine has 4 components

**(i)Interpreter (ii) JIT Compiler (iii) HotSpot profiler (iv) GC**

**1. Interpreter**

It is responsible to read byte code and interpret into machine code (native code) and execute that machine code line by line. The problem with interpreting is it interprets every time even some method invoked multiple times which affects performance of the system. To overcome this problem SUN people introduced JIT compilers in 1.1 V.

**2. JIT Compiler**

The JIT compiler has been introduced to compensate for the disadvantages of the interpreter. The main purpose of the JIT compiler is to improve the performance. Whenever JVM across any method calls, first that method will be interpreted normally by the interpreter but if that method is a repeatedly used (called) method (Hotspot), immediately the JIT compiler compiles that method and generates corresponding native code. Next time JVM comes across that method call then JVM directly uses native code and executes it instead of interpreting once again, so that performance of the system will be improved. Profiler which is the part of JIT compiler is responsible to identify Hotspot (Repeated Used Methods).

**There are other components also inside “Execution Engine”**

**Garbage Collector**

**Security manager**

**If a method is repeatedly invoked methods. Who is going to take care whether it is a repeatedly required method or Hotspot method ?**

Hotspot Profiler is responsible to identify whether a method is repeatedly required ( i.e. it is HotSpot). Sometimes while executing java programs my execution engine may require native method library that’s why we have “Native Method Libraries “ inside JRE.

**Who is responsible for providing “Native Method Libraries” information?**

**Java Native Interface [ JNI ]**

**So Execution Engine will communicate with “JNI”**

**JNI will provide information about “Native Method Libraries” to the Execution Engine.**

1. **What is the job done by the class loader?**

Class loader loads the java class. In java, classes are loaded on demand basis i.e. only when u use the class for the first time in your application, class will be loaded.

1. **Explain the hierarchy of class loaders in java**

**There are 3 types of class loaders in java**

1. **Bootstrap Class Loader or Primordial Class Loader**
2. **Extension Class Loader**
3. **System Class Loader**

**BootStrap or Primordial class loader:**

BootStrap class loader or Premordial class loader loads those classes which are essential for JVM to function properly. BootStrap class loader is responsible for loading all core java classes for instance java.lang.\* ,java.io.\* etc. BootStrap class loader finds these necessary classes from “jdk/jre/lib/rt.jar” .

**Extension class loader:**

Extensions Classloader is the immediate child of Bootstrap classloader. This classloader loads the classes in the lib\ext directory of the JRE.

**System-Classpath class loader:**

It is the immediate child of Extensions classloader. It loads the classes and jars specified by the CLASSPATH environment variable, java.class.path system property, -cp or -classpath command line settings. If any of the jars specified in one of the above manner have a MANIFEST.MF file with a Class-Path attribute, the jars specified by the Class-Path attribute are also loaded. This class loader is also used to load an application’s entry point class that is the “static void main()” method in a class.

1. **What is the role played by Bytecode Verifier?**

Bytecode Verifier performs verification of bytecodes. When a class loader presents the bytecodes of a newly loaded Java platform class to the virtual machine, these bytecodes are first inspected by a verifier. The verifier checks that the instructions cannot perform actions that are obviously damaging. All classes except for system classes are verified.

1. **What are the memory areas allocated by JVM?**
2. **Class(Method) Area or metaspace(according to java8):**

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

1. **Heap:** It is the runtime data area in which objects are allocated
2. **Stack:** Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as the thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

**d) Program Counter Register:**

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

**e) Native Method Stack:**

It contains all the native methods used in the application.

1. **What is the difference between Java compiler ( javac ) and JIT ?**

javac converts source code into the bytecode whereas JIT converts bytecode into native code.

1. **Is JRE different for different Platforms?**

**Yes**

1. **What is garbage collection?**

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed.

1. **Can we force garbage collection programmatically?**

No, we can at most request for Garbage Collection using “System.gc()” or “Runtime.gc()”.

1. **What is an alternative to the finalize method?**

close() or destroy()

1. **What is the difference between System.gc() and Runtime.getRuntime().gc() ?**

Both are the same. System.gc() is effectively equivalent to Runtime.gc(). System.gc()internally calls Runtime.gc().

The only difference is System.gc() is a class method whereas Runtime.gc() is an instance method. So, System.gc() is more convenient.

If you look at System class source code, it has got following method:

public static void gc()

{

Runtime.getRuntime().gc();

}

1. **When do you get a NullPointerException?**

When a reference variable contains null and if it’s used to invoke a member, we get a **NullPointerException.**

1. **What is the signature of the main function in java?**

The signature of the main method is public static void main(String[] ags).

**Public:** It is an Access modifier, which specifies from where and who can access the method. Making the main() method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class.

**Static:** It is a keyword which when associated with a method, makes it a class related method. The main() method is static so that JVM can invoke it without instantiating the class. This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the main() method by the JVM.

**Void:** It is a keyword and used to specify that a method doesn’t return anything. As the main() method doesn’t return anything, its return type is void. As soon as the main() method terminates, the java program terminates too. Hence, it doesn’t make any sense to return from the main() method as JVM can’t do anything with the return value of it.

**main:** It is the name of Java main method. It is the identifier that the JVM looks for as the starting point of the java program. It’s not a keyword.

**String[] args:** It stores Java command line arguments and is an array of type java.lang.String class. Here, the name of the String array is args but it is not fixed and users can use any name in place of it.

1. **Who invokes the main() function ?**

**JVM**

1. **Can we define more than one public class in a java source code? What is the rule of public class and file name?**

There can be only one public class in a java file because the name of the java file is the same as the name of the public class.

1. **Difference between path and classpath**

PATH is the environment variable where we specify the locations of binaries.

Example: We add bin directory path of JDK or JRE, so that any binaries under the directory can be accessed directly without specifying absolute path.

CLASSPATH is the path for Java applications where the classes you compiled will be available.

1. **What is the rule for local members in java?**

Local members must be initialized before use.

1. **What is the class file known as?**

ByteCode

1. **How is java strong object oriented?**

Java does not allow global members, friend functions. Everything (variables as well as methods) must be defined inside the class.

1. **What is the finalize() method ?**

Finalize method in java is called before Garbage collector reclaim the Object. It’s last chance for any object to perform cleanup activity i.e. releasing any system resources held, closing connection if open etc. Main issue with the finalize method in java is that it is not guaranteed by JLS that it will be called by Garbage collector or exactly when it will be called, for example an object may wait indefinitely after becoming eligible for garbage collection and before its finalize() method gets called. Because of this reason it makes no sense to use the finalize method for releasing critical resources. It may work in development in one JVM but may not work in other JVM.

1. **Why is there no sizeof operator in java?**

Because the size of all the java data types is identical across all platforms.

1. **What kinds of programs can we develop using Java?**

There are mainly 5 types of applications that can be created using java:

**a) Standalone Application**

It is also known as a desktop application or window-based application. An application that contains the “main” function as an entry point.

**b) Applet**

It is a java program which can run on java-enabled web browser

**c) Web Application**

An application that runs on the server side and creates dynamic pages, is called a web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

**d) Enterprise Application**

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

**e) Mobile Application**

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

1. **You have a reference type as a member of class What is the default value it gets?**

null

1. **What are the expressions allowed in the switch block of java?**

byte,short,int,char and String (JDK 1.7 onwards…)

1. **In a java source code there are 3 classes defined.When you compile the source code how many class files u get and what are their names?**

**class A {**

**}**

**class B {**

**}**

**class C {**

**}**

A.class, B.class, C.class

1. **What is Java's old name?**

Oak

1. **What technologies come under the “Distributed Object System “?**

**RMI (Remote Method Invocation)**

1. **What do you mean by Java Enabled Browser, Web Server and Application Server?**

The main difference between Web server and application server is that web server is meant to serve static pages e.g. HTML and CSS, while Application Server is responsible for generating dynamic content by executing server side code e.g. JSP, Servlet or EJB.

1. **What is “this”? What is its importance?**

“this” is a reference to the current object or invoking object. With the help of “this” reference member function comes to know about the invoking object. Actually when you invoke any member function, the compiler implicitly passes the “this” reference to the method being invoked.

1. **What is the UnsupportedClassVersion error?**

When a upper-version class file is executed in the lower-version JVM, (e.g. javac belongs to 8 and JRE belongs to 1.7 )

1. **What is a Unicode Character Set in Java?**

Since Java is designed to support Internationalization ( I18N ), it makes sense that it would use Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets such as Latin, Greek, Arabic and many more. For this purpose, it requires 16 bits. Thus char in java is 16 bit type. The range of char is 0 to 65535. There are no negative chars.

1. **What is the difference between Lazy and Eager Resolution?**

Suppose the Car class has Engine as a member.

Here if Car has Engine as static member, Engine will be resolved (loaded) as soon as Car is loaded (Eager). On the other hand, if Car has Engine as a non-static member, Engine will be resolved (loaded) only after you instantiate the Car (Lazy).

1. **Does Java pass by reference or by value?**

Java passes everything by value, and not by reference. When we say everything, that means – Primitive types as well as reference types, these are all passed by value in Java. In case of reference types, what happens is you pass a reference to an object to a method. Inside the method it is collected in another reference (formal argument) to that same object. Now you can modify this formal argument inside the method as much as you want but no matter how hard you try you'll never be able to modify the passed reference that will keep referencing the same Object no matter what!

1. **What is an instanceof operator? give an example**

In Java, instanceof operator is used to check “is-a” relationship at runtime.

#### Example

class FourWheeler{ }

class Car extends FourWheeler { }

public class MyClass

{

public static void main(String[] args)

{

FourWheeler f=new Car();

System.out.println(f instanceof Car); // true

System.out.println(f instanceof FourWheeler); //true

System.out.println(f instanceof Object); // true

}

}

1. **What is upcasting and downcasting with examples?**

**Upcast**: Making base class reference referring to sub class object or assigning subclass object to base class reference.

It is required to make polymorphic calls with an enormous flexibility.

e.g.

Weapon

attack

Gun Sword Riffle

All these child classes have overridden the “attack()” method. We can make a Weapon class reference and make it refer to any child class (existing or may be introduced later) and invoke “attack()” method polymorphically. Here code which invokes “attack()” with the help of “Weapon” class remains unchanged irrespective of changes happening inside inheritance hierarchy [ i.e. any child class gets removed or added ].

**Downcast**: Converting base class reference to subclass reference. It is used to invoke that method of a subclass which is not a part of the base class.

e.g.

In the above hierarchy suppose we have an additional method “fillBullets()” inside Gun. Now when Weapon reference refers to Gun, we would like to invoke “fillBullets()” along with “attack()”. For this reason we need to downcast from Weapon to Gun.

1. **What is the risk involved in downcasting? How does the instanceof operator provide a solution for that?**

void perform(Weapon ref) {

Gun g=(Gun)ref;

g.fillBullets();

g.attack();

}

If someone calls the “perform()” method by passing the “Sword” instance, we will get “ClassCastException”. We can improve our code as follows:

void perform(Weapon ref){

if(ref instanceof Gun)

{

Gun g=(Gun)ref;

g.fillBullets();

}

ref.attack();

}

In case of the above code we downcast from Weapon to Gun only if ref refers to Gun. Thus “instanceof” saves us from getting “ClassCastException”.

1. **In Java copy constructor is not provided implicitly Can we provide it explicitly and if yes when exactly it will be called?**

Yes we can provide copy constructor explicitly in java as follows:

Suppose class name is “MyClass” then the copy constructor would be:

MyClass(MyClass ref)

{

…… code for copy…

}

So if you say….

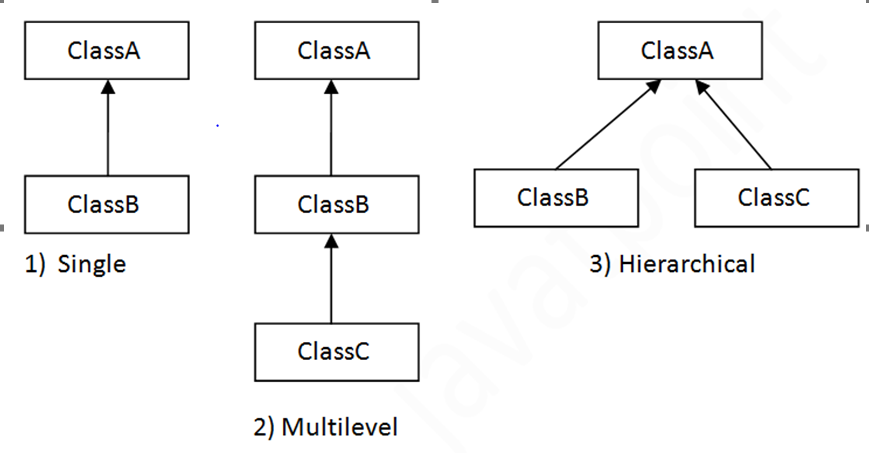
MyClass m1=new MyClass();

MyClass m2=new MyClass(m1);

Copy constructor would be invoked for the second object.

1. **How many types of inheritance are allowed in java?**

**Three types of inheritance: single, multilevel and hierarchical.**

****

1. **Why is “Multiple Inheritance” not supported in Java?**

Multiple inheritance permits you to derive a subclass from more than one direct superclass. This poses a problem if two direct parent classes have conflicting implementations from GrandParent. Now which one to follow in the child class? i.e. a famous “*Diamond Problem ''*!. This problem ideally can be solved by declaring a GrandParent class as “virtual base” class, just to ensure that only one copy of GrandParent in the child class object. But these all add complications to the language and hence Multiple Inheritance is not allowed in java.

1. **Explain accessibility modifiers in java**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Accessibility modifiers | Same class | Sub class in same package | Different class in same package | Sub class in different package | Different class in different package |
| Private | Yes | No | No | No | no |
| <default> | Yes | Yes | Yes | No | no |
| Protected | Yes | Yes | Yes | Yes | no |
| Public | Yes | Yes | Yes | Yes | yes |

1. **What is a super keyword?**

Super keyword is used to access parent class members ( member functions as well as constructors). When super keyword is used to invoke parent class constructor, it has to be on the first line of child class constructor.

1. **What is @Override? What does it do?**

@Override is an annotation which instructs the compiler that we are overriding a particular method and then the compiler will ensure that you follow all the rules while overriding especially method name, parameters are correct etc.

1. **What are Overriding rules in java**

There are 4 rules for overriding in java.

1) Arguments must be the same otherwise it becomes "overloading".

2) Return type of the overriding method can be co-variant.

3) Overriding method must have the same or more accessibility as compared to the overridden method.

4) Overriding and checked-exception rule:

a) Overriding method may not declare any checked exception.

b) Overriding method can declare same checked exception or its sub-type declared by overridden method.

c) Overriding method cannot declare checked exception not declared by overridden method.

1. **What is Object class?**

Object class is the parent class of all the classes in java.

1. **What is “ClassCastException”? What is the solution for that?**

When you try to cast the classes out of hierarchy, you get a ClassCastException.

**Solution:**- before casting check the hierarchy with “instanceof” operator

1. **Where can we apply the final keyword in java?**

Final keyword can be applied to

a) instance member

b) class variable

c) local variable

d) member function

e) class

1. **Can a class be defined as both final and abstract?**

No, because final and abstract are contradictory. i.e. abstract encourages inheritance whereas final stops inheritance.

1. **Explain “System.out.println”**

class System

{

public static PrintStream out;

public static InputStream in;

}

class PrintStream

{

public void print(){}

public void println(){}

}

println() is a non-static method of the PrintStream class, hence it has to be invoked using the reference of PrintStream.

out is a reference to PrintStream. i.e. out.println()

out is a public static member of System class .

that’s why “System.out.println()”

1. **Can we use “this” with static members?**

No. “this” and “static” both are contradictory. i.e. this is a reference to invoking object and static is not at all related to object.

1. **Explain static block or static initializer**

**Static block**

a) is used to access static members.

b) is called as soon as class gets loaded.

c) you can define more than one static block inside the class. They will get executed in the order in which they are defined.

1. **Where have you seen an application of static block or initializer in java?**

In JDBC all the drivers are java classes. *Every jdbc driver class contain static initializer or static block. Inside this static block driver class tries to register itself with DriverManager*. Advantage of this is that the user does not need to explicitly invoke the “registerDriver” method of DriverManager. He just has to ensure that the driver class should be loaded and everything else happens automatically.

1. **What is a non-static block?**

**Non-static block is used if we have many constructor inside a class and those constructors need to have some common statements.** Instead of repeating those statements in each constructor, we place those statements in non-static blocks.

e.g counter which is incremented in each constructor , to keep a track of the number of objects created.

1. **Can static member functions access non-static data?**

No. because static member functions can be called without creating objects. When an object is not created, a non-static member is not allocated memory.

1. **Can non-static member functions access static data?**

Yes. Because in order to invoke non-static member function you need to create object and by that time static members are already allocated memory

1. **Explain constructor invocation in case of inheritance**

When you instantiate a child class, the child class constructor by default invokes “no-arg” constructor of parent class. If parent class does not have “no-arg” constructor, then child class must invoke one of the available constructors of parent class explicitly ( using “super”)

1. **In case of inheritance, if the parent class does not have a no-arg constructor, what happens? How do you solve the problem?**

In such a case, when you try to instantiate a child class you get a compile-time error. Solution:- If parent class does not have “no-arg” constructor, then child class must invoke one of the available constructors of parent class explicitly ( using “super”).

1. **In case of an array, which variable is created automatically? Can you modify it? If not, why?**

In Java, whenever you create an array, you get a “length” variable by default which is nothing but the size of the array. No, you can't modify it because it is final.

1. **What is class Class in java?**

In java whenever any class gets loaded, it is represented by an instance of class Class. All instances of a particular class share the same Class object and you can obtain the Class object by calling the getClass() method of the object. This method is inherited from java.lang.Object

Suppose you create two instances of class called Person e.g.

Person A = new Person();

Person B = new Person();

if(A.getClass() == B.getClass())

{

System.out.println("A and B are instances of same class");

}

Else

{

System.out.println("A and B are instances of different class");

}

In this case it will print "A and B are instances of the same class" because they are both instances of class Person.

1. **What is the use of class “Class” in java?**

The main use of class “Class” is in the case of Reflection API, the technique to find out everything about a particular class during runtime.

1. **What does the “forName() “ function do ?**

It is used to load a particular class explicitly. It returns the Class object associated with the class or interface loaded.

1. **When does class get loaded?**

a) implicitly

java MyClass

or

emp e=new emp(); when u create first object of that class

or

emp.staticmemberfun();

Whichever is first……..

b) Explicitly

Class.forName("classname");

1. **What is the bridge method in java?**

When you override the parent class method with a covariant return type, the compiler synthesizes (implicitly provides) additional method in child class with a matching signature of the overridden method. This method invokes the method which we are overriding. This synthesized method is known as “Bridge Method ''.

1. **Why does the compiler synthesize the bridge method in case of overriding?**

Suppose we have overridden a parent class method with covariant return type e.g.

class base

{

public Object disp()

{

System.out.println("base disp");

return null;

}

}

class sub extends base

{

public String disp()

{

System.out.println("sub disp");

return null;

}

}

Even though the above overriding is correct according to the compiler, for JVM this is not exactly an overriding because their return types are different. To solve this problem, the compiler adds a bridge method [public Object disp( ) ] in Derived class [sub] which calls the method which we have overridden.

1. **What are the scenarios in which we have early binding in java?**

In case of private or static or final methods.

1. **What is the .length variable in case of array? Can we see it using the Reflection API technique?**

When you create any type of array internally, a length variable gets created which stores the size of array. Its signature is “public final int length. Arrays are not like regular classes, they are a special case defined in the [Java Language Specification](http://java.sun.com/docs/books/jls/second_edition/html/arrays.doc.html). So the .length public final variable is defined in the Java Language Specification, but it is not actually defined anywhere in a .java or a .class file. Hence we cannot see it using Reflection API technique

1. **Explain abstract class in Java** 
   1. In Java, as soon as you define a class with an "abstract" keyword, the class becomes abstract.
   2. Abstract class cannot be instantiated.
   3. Abstract classes can contain abstract as well as non-abstract methods.
   4. Abstract method is a method which is declared with the "abstract" keyword.
   5. A child class of an abstract class has to provide implementation to the method which is declared "abstract" in parent class or else make the child class also "abstract".
2. **Explain interface in Java8**

Interfaces are abstract in nature. i.e. they can not be instantiated.

Interface can contain

* members
* abstract methods
* default methods
* static methods

Members declared in the interface are by default "public","static" and "final".

Methods declared (and not defined) inside the interface are by default "public" and "abstract".

A class can be derived from more than one interface (implements keyword is used here) child class of an interface has to provide implementation of the method/s which are declared abstract in parent or else child class also has to be declared as "abstract".

Default method/s (which are by default "public") may or may not be overridden by child class. ( if overridden, "public" modifier is compulsory)

static methods are like utility methods which can be invoked only on the interface in which they are defined.

1. **Why do we need an interface?**

Interfaces are important because they separate what a class does from how it does it. The contract defining what a client can expect leaves the developer free to implement it any way they choose, as long as they uphold the contract.

You see examples of this all over the JDK. Look at the java.sql package - it's riddled with interfaces. Why? So different relational databases can be free to implement those methods for their particular product. Clients need only deal with the interface reference types. Changing relational databases is as easy as swapping one JDBC driver JAR for another. Clients need not change. (As long as they don't stray from the contract.

You can program to an interface instead of a concrete implementation in order to make your code flexible.

In addition to this, interfaces make unrelated classes relate e.g.

class Aeroplane extends Vehicle{}

class SuperMan extends Person{}

class Bird extends Animal{}

Though these classes are unique, there exists one common factor “fly()”

So we can relate these unrelated classes with the interface “Flyable”

interface Flyable

{

void fly();

}

class AeroPlane extends Vehicle implements Flyable{public void fly(){ fly like AeroPlane}}

class SuperMan extends Person implements Flyable{public void fly(){ fly like SuperMan}}

class Bird extends Animal implements Flyable{public void fly(){ fly like Bird}}

And now somewhere we can have code :

void perform(Flyable ref)

{

ref.fly();

}

And we can pass :

perform(new AeroPlane());

perform(new SuperMan());

perform(new Bird());

1. **Which of the below invocations are illegal at compile-time?**

**interface myinterface**

**{ void disp(); }**

**public class MyClass**

**{**

**public static void main(String[] args)**

**{**

**myinterface m1,m2=null;**

**m1.disp();**

**m1.fun();**

**m1.toString();**

**m1.equals(m2);**

**m1.hashCode();**

**}**

**}**

only “m1.fun();” is illegal, the rest all are legal at compile-time. It is true that using interface reference we can invoke only those methods which are present in it. But “toString()” ,”equals()” and “hashCode()” invocations are legal because these methods are available inside “java.lang.Object” which is the base class of all the classes in java. Now the compiler understands that any implementation of a given interface is by default a subclass of “java.lang.Object”, hence the compiler allows you to invoke above mentioned methods using interface reference.

1. **When to use interface and when to use abstract class? Explain with example**

**For Interface:**

Interface is used when you don't know anything about implementation but know the contract that implementer should have to accomplish the task.

**For Abstract class:**

Abstract class is used when you know partial implementation, where say out of 5 methods, you know implementation of 3 methods and don't know implementation of 2 methods in that case 2 methods will be abstract and you need to rely on implementer as a contract to must provide body of abstract methods to accomplish the task.

1. **How many ways can we achieve Polymorphism in Java?**

**Polymorphism using class**

class Animal

{

int legs;

void makeSound()

{

Cout<<”Groul”;

}

}

Class Cat extends Animal

{

void makeSound()

{

Cout<<”meow…”;

}

}

Animal a=new Cat();

a.makeSound();

**Polymorphism using interface.**

interfaces make unrelated classes relate e.g.

class Aeroplane extends Vehicle{}

class SuperMan extends Person{}

class Bird extends Animal{}

Though these classes are unique, there exists one common factor “fly()”

So we can relate these unrelated classes with the interface “Flyable”

interface Flyable

{

void fly();

}

class AeroPlane extends Vehicle implements Flyable{public void fly(){ fly like AeroPlane}}

class SuperMan extends Person implements Flyable{public void fly(){ fly like SuperMan}}

class Bird extends Animal implements Flyable{public void fly(){ fly like Bird}}

And now somewhere we can have code :

void perform(Flyable ref)

{

ref.fly();

}

And we can pass :

perform(new AeroPlane());

perform(new SuperMan());

perform(new Bird());

1. **What is the difference between abstract class and interface in Java8?**

Abstract class can define constructor. They can have a state (instance members) associated with them. Interfaces still can't have any state. Interfaces still can't have any final method, which means that any implementation can override all its default methods. And interfaces still can't have any constructor.

You can still implement multiple interfaces, even if they have default methods with the same signature. You can't extend multiple classes (abstract or not).

1. a class may inherit from only one other class, but can implement many interfaces.

2. an interface may not have any fields, except defining constants, while an abstract class can.

3. an abstract class may define a constructor, while an interface cannot.

1. **What are the default methods in Java8 interfaces?**

A *default method* is an instance method defined in an interface whose method header begins with the default keyword; it also provides a code body. Every class that implements the interface inherits the interface's default methods and can override them

1. **What is the actual use of the “default” method in the interface?**

By default, interfaces are too tightly coupled with their implementation classes. I.e., it is not possible to add a method in the interface without breaking the implementer class. Once you add a method in the interface, all its implemented classes must declare the method body of this new method. e.g., if you add a single method in interface List, it breaks everything. You need to add its implementation in every class that implements List interface. Imagine in the real world how many custom classes would change.

So for backward compatibility, Java 8 cleverly added Default Methods.

**Virtual Extension Methods**, or as they are often called defender methods, can now be added to interfaces providing a default implementation of the declared behaviour. So existing interfaces can be enhanced without compromising backward compatibility by adding extension methods to the interface.

One of the major reasons for introducing default methods is to enhance the Collections API in Java 8 to support lambda expressions.

A key characteristic of extension methods is that they are virtual methods just like other interface methods, but provide a default implementation in the event that the implementing class does not provide a method body.

1. **What do you mean by the “Class wins” rule in java8?**

If, class C extends A implements B

Now class A has a concrete method “void disp()” and interface B has a default method “void disp()”. In this case C will inherit the method from class A and the default method from interface B will simply be ignored. This is the “Class wins” rule in java8. The “Class wins” rule ensures compatibility with Java7.

1. **Why is it not recommended to make a default method that redefines one of the methods from the Object class?**

As a consequence of the “Class wins” rule, such a method could never win against Object.toString() or Object.equals().

1. **Why do we need static methods in the interface?**

Static methods are the utility methods. Before Java8, it was common to place static methods in companion classes. You find pairs of interfaces and utility classes such as Collection/Collections or Path/Paths in the standard library. With the introduction of static methods now we don’t require such utility classes.

1. **Explain this vs super in case of constructor invocation**

this is used to invoke the constructors of the same class (constructor chaining ) whereas super is used to invoke parent class constructor.

1. **What is equals and hashcode contract?**

a) Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.

b)If two objects are equal according to the equals(Object) method, thencalling the hashCode method on each of the two objects must produce the same integer result.

*c)* It’s not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

1. **What is the difference between == and equals() method ?**

Both == operator and equals() methods are the same i.e. they check whether two references refer to the same object or not. This is because java.lang.Object class's "equals()" method internally uses == operator. Now the difference is that "equals()" method can be overridden in order to check the content i.e. values of instance members but since java doesn't support operator overloading we cant overload == operator.

1. **Explain toString, equals and hashCode methods of Object class**

**public String toString()**

returns the String representation of an instance.

**public boolean equals(Object)**

checks the equality of two references. If they are referring to the same instance then they are equal otherwise not.

**public int hashCode()**

every object is given a unique number inside memory. This number is called a hashcode. This method returns the hashcode of the caller object.

1. **What do you mean by immutable class?**

When you perform any operation on any instance, it will not affect the same one, rather it will create a new one with modifications

1. **How to create an immutable class in java?**

In order to create immutable class , we need to follow some guidelines:

a) Don't provide "setter" methods — methods that modify fields

b) If the instance fields include references to mutable objects, don't allow those objects to be changed:

i.e. don’t provide methods that modify the mutable objects.

c) If the instance fields include references to mutable objects, don't allow those objects to be changed:

Don't share references to the mutable objects.

d) If necessary, create copies to avoid returning the originals in your methods

e) A class should be final

1. **What are wrapper classes? What is their main application?**

Wrapper classes are used to convert any primitive type into an object. The primitive data types are not objects, they do not belong to any class, they are defined in the language itself. While storing in data structures which support only objects, it is required to convert the primitive type to object first, so we go for a wrapper class.

1. **JDK 15 onwards, we can pass primitives (ie int,char,float etc) Directly to the method accepting “javalangObject” Does it mean that there is no use of Wrapper classes now?**

No we can’t say that. It is true that JDK 1.5 onwards, we can pass primitives (i.e. int,char,float etc.) Directly to the method accepting “java.lang.Object”. But what actually happens internally is, primitives are converted to Wrapper (autoboxing) and then Wrapper being a subclass of “java.lang.Object” can be passed to the method accepting “java.lang.Object”. (widening)

1. **What is autoboxing and unboxing?**

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and the opposite operation is known as unboxing.

1. **Difference between StringBuilder and StringBuffer**

StringBuilder is not thread-safe whereas StringBuffer is thread-safe (all of its methods are synchronized)

1. **Autoboxing and unboxing are limited to compilers only. What do you mean by this?**

All the autoboxing and unboxing instructions u write in your source code are removed by the compiler while compiling the code. JVM does not understand "autoboxing" and "unboxing". Hence the compiler removes them while compiling the source code.

1. **What technique is used to make a variable number of arguments possible in java?**

Compiler automatically converts variable number of arguments to array.

1. **When you try to print any reference variable in java, you get some meaningful output. What do you mean by this?**

It means that the class must have overridden the “toString()” method.

1. **Why can’t we instantiate abstract class?**

An Abstract class represents an abstract concept. Take an example of vehicle class. You cannot build a vehicle that is not something more specific. An abstract class is not complete! The author marked it abstract to tell you that some implementation is missing in the code. The author has done some of the work, but you must fill in some bits yourself to get it working. The abstract keyword ensures that no one would accidentally initiate this incomplete class.

1. **Can abstract classes have constructors in Java?**

Yes, abstract classes can have constructor/s in Java. Since you cannot create instances of abstract class, constructor can only be called during constructor chaining, i.e. when you create an instance of concrete implementation class. Now the question arises what is the purpose of constructor, if you cannot instantiate abstract class? Well, it can still be used to initialize instance members, which are declared inside abstract classes, and used by various implementations. Also, even if you don’t provide any constructor, the compiler will add default no argument constructor in abstract class, without that your subclass will not compile, since the first statement in any constructor implicitly calls super (), default super class constructor in Java.

1. **Can abstract classes implement interfaces in Java? Do they require all methods?**

Yes, abstract classes can implement interfaces by using implements keyword. Since they are abstract, they don’t need to implement all methods. It’s good practice to provide an abstract base class, along with an interface to declare Type. One example of this is java.util.List interface and corresponding java.util.AbstractList abstract class. Since AbstractList implements all common methods, concrete implementations like LinkedList and ArrayList are free from the burden of implementing all methods, had they implemented List interface directly. It’s best of both world, you can get advantage of interface for declaring type, and flexibility of abstract class to implement

1. **Can abstract classes be final in Java?**

No, abstract classes cannot be final in Java. Making them final will stop abstract class from being extended, which is the only way to use abstract class. They are also opposite of each other, abstract keyword enforces to extend a class, for using it, on the other hand, final keyword prevents a class from being extended. In the real world also, abstract signifies incompleteness, while final is used to demonstrate completeness. Bottom line is, you cannot make your class abstract and final in Java, at same time, it’s a compile time error.

1. **Can abstract classes have static methods in Java?**

Yes, abstract classes can have static methods.

1. **Is it necessary for abstract classes to have abstract methods?**

No, It’s not mandatory for an abstract class to have any abstract method. You can make a class abstract in Java, by just using the abstract keyword in class declaration. Compiler will enforce all structural restrictions, applied to abstract classes, e.g. not allowing to create any instance. By the way, it’s debatable whether you should have an abstract method inside an abstract class or interface.

1. **Can abstract classes contain the main method in Java?**

Yes, abstract class can contain the main method, it is just another static method and you can execute the Abstract class with the main method, until you don’t create any instance.

1. **What do you mean by concrete class?**

A class which is not an abstract class i.e. a class which is a complete class ,which can be instantiated is known as a “concrete class”.

1. **What do you mean by “Functional Interface”?**

“Functional Interface” is the one having only one abstract method also known as “SAM (Single Abstract Method). It may have default or static methods.

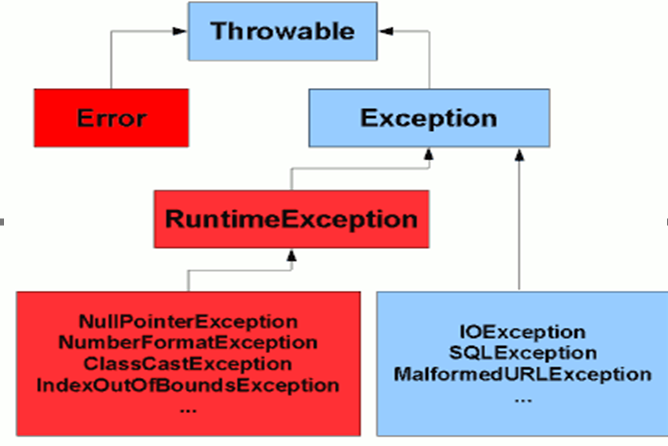
1. **What Is an Exception?**

An exception is an event, which occurs during the execution of a program, that interrupts the normal flow of the program's instructions.

1. **Why do we need Exceptions?**

* In traditional programming, error detection, reporting, and handling often lead to confusing code because programmers would use error code inside the main logic. Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere.
* A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. i.e. if a method does not want to handle an exception it can propagate it to the caller and so on. we did not have this advantage in traditional programming.
* Third advantage is, because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

1. **Describe the exception hierarchy in java**

****

1. **What is the difference between “Exception” and “Error”?**

**Both Error and Exception are derived from java.lang.Throwable.**

Examples of Error are:

**java.lang.OutOfMemoryError or Java.lang.NoClassDefFoundError java.lang.UnSupportedClassVersionError etc.**

examples of Exception are:

**java.lang.RuntimeException (unchecked) , java.io.IOException (checked) etc.**

The first difference is Error is unchecked by nature whereas Exceptions can be checked or unchecked.

The second difference is that Error is not meant to catch as even if you catch it you cannot recover from it. For example during OutOfMemoryError, if you catch it you will get it again because GC may not be able to free memory in the first place. On the other hand, exceptions can be caught and handled properly.

i.e. Errors are fatal in nature and recovery may not be possible, on the other hand by carefully handling Exceptions you can make your code more robust and guard against different scenarios.

An exception is an event that represents a condition from which it is possible to recover, whereas error represents an external situation usually impossible to recover from. All errors thrown by the JVM are instances of Error or one of its subclasses, the more common ones include but are not limited to:

* OutOfMemoryError – thrown when the JVM cannot allocate more objects because it is out memory, and the garbage collector was unable to make more available
* StackOverflowError – occurs when the stack space for a thread has run out, typically because an application recurses too deeply
* ExceptionInInitializerError – signals that an unexpected exception occurred during the evaluation of a static initializer. E.g.

1. **What happens when an exception is raised in a program?**

When an exception occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack.*

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the program execution aborts.

1. **Why do you handle the exception?**

Exception handling ensures normal program execution even after an exception gets raised.

1. **How to handle the exception?**

using try….catch block

1. **When you write one try and multiple catches, what care should you take?**

The most specific catch block should precede the most generic catch block.

1. **Can we write catch block in a following manner: catch(Object ref){ }**

No, we cannot write catch blocks that way. **Only the “Throwable” class or one of its child classes can be the argument type in a catch clause.** It is because, if the catch block is allowed to be written in this way, we can use “throw new String()” or “throw new Thread()” etc.

All the above-mentioned combinations are illegal because **Objects that are instances of “Throwable” class (or one of its child classes) only are thrown by the JVM or can be thrown by the Java throw statement.**

1. **What is the “handle or declare” rule?**

a) Whenever any method raises checked exception/s , the method has to either handle [try….catch] or declare [throws] that checked exception/s.

b) Whenever you invoke a method which has declared [using throws] checked exception/s , the caller method has to either handle [try….catch] or declare [throws] that checked exception/s.

1. **What is the finally block? Why do we need it?**

finally block is the one which gets executed irrespective of whether exception is raised or not. Since it is guaranteed to be invoked unlike the finalized () method, we can release the resources such as file, database connection, sockets etc. inside it.

1. **In which scenarios the finally block will not get executed?**

System.exit(0) in try or catch and any unhandled exception is raised inside the finally block.

1. **Define checked and unchecked exception**

Checked exceptions: Checked exceptions are those exceptions which can be raised in a correct program. Client code has to either handle the checked exceptions or declare them with the throws clause.

Unchecked exceptions: Unchecked exceptions are those exceptions which can be raised due to programming mistakes. Unchecked exceptions derived from RuntimeException. Exceptions that inherit from RuntimeException get special treatment. There is no requirement for the client code to deal with them

1. **In java, adjectives ending in -able are interfaces Serializable, Comparable , Cloneable etc So why is Throwable a class and not an interface?**

It is because they wanted to have some state associated with every exception that gets thrown such as message, cause, and stack trace etc. and you can't do that with interfaces**.**

1. **When will you create a checked exception?**

When you realize that client code will take some useful recovery action based on information in exception You can go for checked exception.

1. **How do you create checked and unchecked exceptions?**

We create checked exceptions by deriving from “Exception” class whereas we create unchecked exceptions by deriving from “RuntimeException” class.

1. **Compiler enforces you to deal with checked exceptions? What do you mean by this?**

You must either handle ( using try… catch) or declare (using throws)

1. **Difference between throw and throws**

throw is used to raise the exception whereas throws is used to declare the exception.

1. **What is an ARM block?**

ARM (Automatic Resource Management) block is also known as “try with resource” block.

e..g try(FileInputStream fis=new FileInputStream())

{

Some statements….

}

catch(FileNotFoundException fe)

{

fe.printStackTrace();

}

The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.io.Closeable, can be used as a resource.

1. **Is it recommended to handle “NullPointerException”,”ClassCastException” etc using java code?**

No, not at all. Exceptions are costly, and can slow down your code. Don’t just throw and catch exceptions, if you can use if..else expression to indicate result of operation, e.g. NullPointerException, ClassCastException etc. which may result in cleaner code and performance solution

1. **What is Re-throwing an exception in java?**

Exceptions raised in the try block are handled in the catch block. If it is unable to handle that exception, it can re-throw that exception using throw keyword. It is called re-throwing an exception.

try

{

String s = null;

System.out.println(s.length()); //This statement throws NullPointerException

}

catch(NullPointerException ex)

{

System.out.println("NullPointerException is caught here");

throw ex; //Re-throwing NullPointerException

}

1. **What are the legal combinations of try, catch and finally blocks?**

**Ans:-**

A)

try

{

//try block

}

catch(Exception ex)

{

//catch block

}

B)

try

{

//try block

}

finally

{

//finally block

}

C)

try

{

//try block

}

catch(Exception ex)

{

//catch block

}

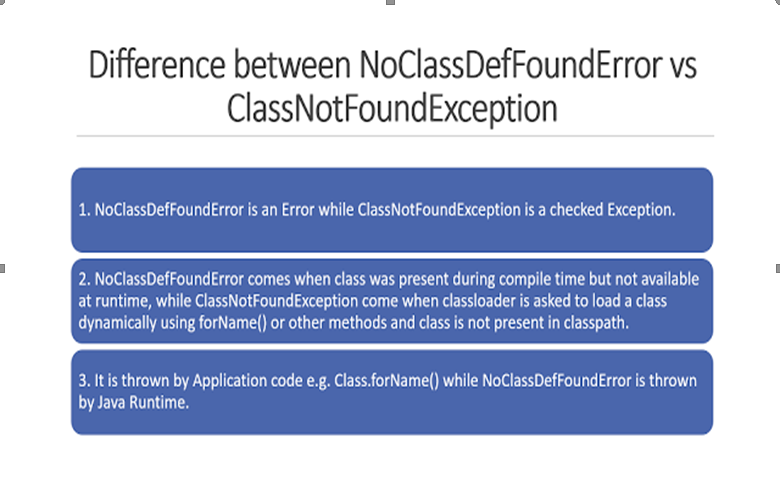
finally

{

//finally block

}

1. **When deciding on checked exceptions vs unchecked exceptions, ask yourself, "What action can the client code take when the exception occurs?"**
2. **What is the difference between NoClassDefFoundError and ClassNotFoundException?**

****

1. **What are the performance implications about exception handling?**

Exceptions come with a price, and in order to understand some of the issues involved, let's look at the mechanism for handling exceptions. The Java Virtual Machine maintains a method-invocation stack (or call stack) that features a list of the methods that have been invoked by a thread, starting with the first method the thread invoked and ending with the current method. A method-invocation stack illustrates the path of method invocations a thread made to arrive at the current method.

So, for the purposes of our comparison, what needs to happen in Case 1? If a NoPassengerFoundException throws while executing the searchPassengerFlightRecord() method (the top stack frame), further execution of code halts and the Java VM receives control to implement Java's exception-handling mechanism. The Java VM then searches in the current method for a catch clause having NoPassengerFoundException. If it doesn't find such a clause, then the Java VM pops the stack frame for the current method, and the calling method becomes the current method. The Java VM again searches in the current method -- for a suitable catch clause. If the Java VM still doesn't find a suitable catch clause, it pops the current stack frame and then searches the next stack frame for a suitable catch clause, and so on, until it finally finds an appropriate one. Or, if it doesn’t find, then the application terminates with the stack trace printing on the console. Abrupt method completion is significantly more expensive (performance-wise) than a normal method completion

1. **How do you convert checked exceptions to unchecked exceptions?**

try

{

Some statements

…………

}

catch(CheckedException ce)

{

throw new UncheckedException();

}

Now your caller need not handle or declare exception because you have converted checked exception to unchecked exception.

1. **What is the process?**

Process is a program in execution.

1. **What is thread?**

Thread is one of the execution paths within a process.

1. **What is the meaning of Process-based multitasking and Thread-based multitasking? Give examples**

**Process-based multitasking: -** More than one processes are running simultaneously. word and excel applications are running simultaneously.

**Thread-based multitasking: -** More than one threads are running simultaneously. e.g., within a word application, you can start formatting as well as printing.

1. **What do you mean by context switching?**

It means the CPU jumps from one task to another and vice-versa.

1. **Difference between Process-based multitasking & Thread-based multitasking**

a) Threads can share the memory, processes cannot.

b) Context switching between threads is relatively cheaper as compared to that between processes.

c) Cost of communication between threads is also low.

(Cheaper or cost is low actually means less number of system resources are used.)

1. **What is one of the main applications of multithreading in java?**

Multithreading can be used to create an effective server side application so that it can receive multiple clients at a time ( each client will be processed by a new thread ). e.g., Servlet, JSP

1. **What are the main steps in a typical multithreading application?**

a) Create thread/s

b) Define thread execution body

c) Register thread with the thread scheduler

d) Thread scheduler will execute the thread/s

1. **Can we directly call the run() method ? What happens in that case?**

Yes , we can call run() directly. But in that case, it won't be thread execution, it is a normal method call. That is, a different call stack won't be created.

1. **What is the default priority every user thread gets?**

Normal priority

1. **Explain “extends Thread”**

a) Define a class say e.g. MyThread extends Thread

b) Override “public void run()” to define the thread execution body.

c) Create the required number of “MyThread” instances.

d) Invoke “start()” method on the above created instances.

1. **Explain “implements Runnable”**

a) Define a class e.g. MyApp implements Runnable

b) Define “public void run()” to define a thread execution body.

c) Create an instance of “MyApp”

d) Create required number of “Thread” class instances by passing “MyApp” instance as an argument (i.e. target )

e) Invoke “start()” method on the above created “Thread” class instances

1. **In extends Thread, if we don’t override the “run()” method what will happen ?**When scheduler will try to invoke “run()” on a child class, parent class i.e. Thread class run() method will get invoked.
2. **In implements Runnable, if we don’t define “run()” method what will happen**

Compile time error.

1. **What is the Difference between "implements Runnable" and "extends Thread" in Java?**

**1. Inheritance Option:** The limitation with "extends Thread" approach is that if you extend Thread, you cannot extend any other class. Java does not support multiple inheritance. In reality, you do not need Thread class behavior, because in order to use a thread you need to instantiate one anyway. On the other hand, implementing the Runnable interface gives you the choice to extend any class you like, but still define behavior that will be run by a separate thread.

**2. Loosely-coupled:** "implements Runnable" makes the code loosely-coupled and easier to read. Because the code is split into two classes. Thread class for the thread specific code and your Runnable implementation class for your job that should be run by a thread code. "extends Thread” makes the code tightly coupled. Single class contains the thread code as well as the job that needs to be done by the thread.

**3. Functions overhead:** "extends Thread” means inheriting all the functions of the Thread class which we may not need. The job can be done easily by Runnable without the Thread class functions overhead.

1. **You are extending Thread or Implementing Runnable You need to deal with some kind of checked exception inside run() method What will you do? You will declare it or handle it inside run() ? Give reason**

We have to handle it. We cannot declare it as it goes against the overriding and checked exception rule which states that the overriding method cannot declare checked exceptions not declared in the overridden method.

1. **Normally when does thread die?**

When the “run()” method gets over.

1. **Can we restart dead thread?**

No

1. **Why is the start() method not final?**

Because you can override start() and do something useful before or after calling super.start().

1. **What is the main thread?**

main thread is the one created by JVM and it is responsible to invoke main() method.

1. **Explain Race condition with example**

There are 2 threads

one thread is reading from a file

Another thread is writing to a file.

Race condition means

while one thread is reading from a file, another thread might write in a file or vice-versa.

Race conditions always lead to Data Corruption.

1. **How do we avoid the Race condition?**

By using synchronization

1. **What is synchronization?**

Synchronization is a technique used to ensure that when multiple threads are sharing the same resources, there shouldn’t be a “Race Condition” scenario. In other words, synchronization ensures that the lock on an object is acquired by only one thread at a time.

1. **What exactly happens when we use the synchronized keyword?**

There is a concept of object lock. In java every object has a lock. This lock can be accessed by only one thread at a time. The lock will be released as soon as the thread completes its job and thus another thread can acquire the lock. This lock comes into picture only when the object has got non-static synchronized method/s or block. Whichever thread executes the synchronized method first, it acquires the lock. Other thread/s have to be in "seeking lock state". Acquiring and Releasing locks happens automatically. Once a thread acquires a lock on an object, it can have control on all the non-static synchronized methods of that object.

1. **Difference between synchronized method and block**

Synchronized keyword

**method:-** all the statements are protected.

**block:-** only those statements are protected which are given inside the synchronized block.

1. **Once a thread acquires the lock on an object, what are the things it has control on?**

Once a thread acquires the lock on an object, it has control on all the non-static synchronized methods of that object.

1. **What do you mean by “Thread-Safe” class?**

Thread-safe class means a class having non-static synchronized methods or methods having non-static synchronized blocks**.**

1. **What is the drawback of synchronization?**

You get Thread-Safety at the cost of performance.

1. **What is class lock?**

Every class has a lock. It is actually a lock on an instance of class “Class”. This is because, whenever any class is loaded in java, it is represented by an instance of class “Class”. The class lock comes into picture in case of synchronized static methods. Thread which gives a call to synchronized static method can acquire a class lock. Only after thread complete that static method, lock is released

1. **What is the need for Threads Communication?**

Even though synchronized method or block is used to avoid "Race Condition", there can be danger of "Deadlock" inside it.

e.g., if one thread is working inside a synchronized block or method and if it gets stuck up! Imagine what will happen?

Neither this thread can complete and release the lock, nor can another thread acquire the lock. Solution to this is to have a Communication between threads. i.e., if the thread realizes it cannot continue, it should come out of the synchronized method or block and release the lock. Now another thread will acquire the lock, execute the code and allow the first thread to resume.

1. **What are the methods used for Thread communication? Explain them in brief**

In order to have communication between threads java.lang.Object class has 3 methods which are final

wait() , notify(), notifyAll()

a) wait

it will make thread, release the lock and go to the wait pool.

b) notify

it will make the thread move from wait pool to seeking lock state.

c) notifyAll

it will make all the threads move from wait pool to seeking lock state.

1. **If no thread is in the wait pool, and we invoke notify or notifyAll, what will happen?**

nothing will happen. notify() and notifyAll() methods are not having any side effects.

1. **wait(), notify() and notifyAll() methods can be called only from a synchronized method or a block. Why is it so?**

It is because these methods are used to control object lock and object lock comes into picture in case of non-static synchronized method or block.

1. **Difference between sleep and wait**

wait() method releases the lock , whereas sleep does not.

1. **What do you mean by Daemon Thread? Give an example**

Daemon threads are the threads which are at the mercy of user thread/s. Their only purpose is to serve user defined thread/s. When there is no user thread alive, the Daemon thread will die.

e.g. garbage collection thread (low priority thread).

1. **What is InterruptedException?**

Whenever a thread is in a blocked state i.e. due to sleep, join or wait methods, it can get interrupted by other threads. Whenever a blocked thread gets interrupted, it throws an "Interrupted Exception", which is a checked exception.

1. **What do you mean by Thread’s born state?**

It means either you have created an instance of the class which is derived from the Thread class or you have created an instance of Thread class.

1. **What is the difference between runnable and running state?**

When you invoke “start()” method, the thread goes in “runnable” state and when JVM scheduler invokes “run()” method thread goes in “running” state.

1. **From blocked state, thread goes to runnable or running state?**

From blocked state, thread always goes to “runnable” state.

1. **You are using the “implements Runnable” technique for multithreading applications. What happens when you pass an object that implements Runnable to the Thread class constructor and what happens if you don’t pass such object to Thread class constructor?**

Actually when you pass object implements Runnable to Thread class constructor, in Thread class following member is declared:

private Runnable target;

Now when you pass an object that implements Runnable, the above “target” is initialized inside the Thread class constructor.

Thread class has following “run() “ method defined :

@Override

public void run()

{

if (target != null) {

target.run();

}

}

Hence it is quite clear that when you don't pass an object implementing the Runnable to Thread class constructor, “run” method of the Thread class gets invoked and that’s it.

If you pass an object that implements Runnable then, the Thread class run method gets invoked and it invokes “target’s” run method. Here target means “object implements Runnable”.

1. **What is Reentrantlock in java?**

ReentrantLock is a mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended features like fairness, which can be used to provide lock to the longest waiting thread. Lock is acquired by lock() method and held by Thread until a call to unlock() method.

1. **What is the Difference between ReentrantLock and implicit lock?**
2. Synchronized keywords don't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instances of ReentrantLock. Fairness property provides lock to the longest waiting thread, in case of conflict.
3. Second difference between synchronized and Reentrant lock is the tryLock() method. ReentrantLock provides a convenient tryLock() method, which acquires lock only if it's available or not held by any other thread. This reduces [blocking](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html) of threads waiting for lock in Java applications. Similarly tryLock() with timeout can be used to timeout if lock is not available in a certain time period.
4. ReentrantLock also provides a convenient method to get a List of all threads waiting for lock.
5. In case of intrinsic lock if an exception is raised, the lock is automatically released. Whereas in case of Reentrant lock if exception is raised, lock is not released.
6. **What is the drawback of Reentrant lock?**

Programmer is responsible for acquiring and releasing the lock, which is a power but also opens the gate for new subtle bugs, when the programmer forgets to release the lock in the finally block.

1. **What is lock fairness in case of Reentrant lock?**

When the lock is released, waiting threads will acquire it in strict first-come-first-served order. In this mode, it is possible for a thread to have to wait for the lock even though at that precise moment it is not owned by another thread (so that it can go to a "more worthy contender" currently waiting for it).

1. **What is the difference between “tryLock()” and “tryLock(long time, TimeUnit wait)”?**

**boolean tryLock()**

The above method tries to acquire a lock and returns true if it was successful. Otherwise, it immediately returns false, and the thread can go off and do something else.

**boolean tryLock(long time, TimeUnit wait)**

The above method tries to acquire a lock for a given time and returns true if it was successful. If it does not acquire a lock within that time, it returns false.

1. **Explain “Condition Objects” in case of Reentrant lock**

Often, a thread enters a critical section, only to discover that it can’t proceed until a condition is fulfilled. You use a Condition object to manage threads that have acquired a lock but cannot do useful work.

A lock object can have one or more associated condition objects. You obtain a condition object with the “newCondition()” method.

“await()” method of “Condition” , thread is deactivated and gives up the lock.

The thread is not made “runnable” when the lock is available. Instead, it stays deactivated until another thread has called “signal()” or “signalAll()” method on the same condition.

“signal()” method unblocks only a single thread from the wait state, chosen at random.

“signalAll()” method reactivates all threads that are waiting for the condition.

1. **What are Executors?**

Executors are used to implement the Thread Pooling concept.

1. **What do you mean by thread pooling?**

A thread pool contains a number of idle threads that are ready to run. When the run method exits, the thread doesn’t die but stays around to serve the next request. One benefit of a thread pool over creating a new thread for each task is that thread creation and destruction overhead is restricted to the initial creation of the pool, which may result in better performance and better system stability.

1. **How does Executor work?**

You give a Runnable to the Executor, it will create a number of Threads for you, and the scheduler will then invoke run method for those threads.

1. **What is the advantage of Thread pool over creating threads from scratch?**

If requests are frequent and lightweight, as in most server applications, creating a new thread for each request can consume significant computing resources. In this scenario we should instead use a thread pool. A thread pool contains a number of idle threads that are ready to run. You give a Runnable to the pool, scheduler will then invoke run method for those threads. When the run method exits, the thread doesn’t die but stays around to serve the next request.

1. **What do you mean by serialization and deserialization?**

**Serialization** is the process of converting an object's state (including its entire graph) to a sequence of bytes which can be persisted into disk or sent over network to any other running machine. The process of rebuilding those bytes into a live object at some future time is known as deserialization.

1. **What is the need of serialization?**

When you create an object, it exists for as long as you need it, but under no circumstances does it exist when the program terminates. While this makes sense at first, there are situations in which it would be incredibly useful if an object could exist and hold its information even while the program wasn’t running. Then the next time you started the program, the object would be there and it would have the same information it had, the previous time the program was running.

1. **Object serialization was added to java in order to support which two features?**

Object serialization was added to the language to support two major features. Java’s RMI and Java Beans.

When JavaBean is used, it’s state information is generally configured at design time. This state information must be stored and later recovered when the program is started. Object serialization performs this task.

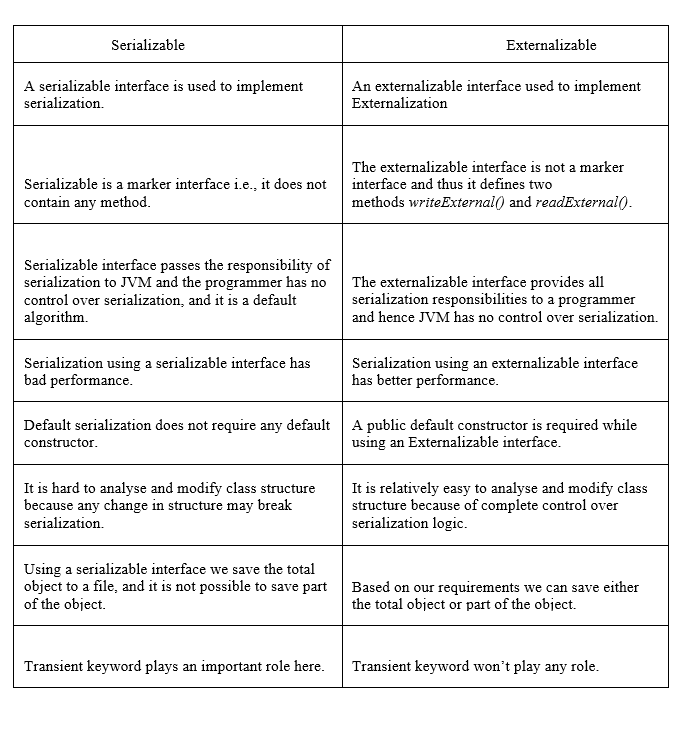
In the case of RMI, when an object is sent over a network, its copy is sent through a serialization mechanism.

1. **What are the requirements we need to meet in order to serialize an object?**

a) class should implement either Serializable or Externalizable interface

b) class non-static members should be of serialized nature.

1. **What is the difference between Serializable and Externalizable?**

****

1. **What is SerialVersionUID in case of Serialization? What happens when we deserialize an object?**

As per java docs, during serialization, runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. Simply put, the serialVersionUID is a unique identifier for Serializable classes. This is used during the deserialization of an object, to ensure that a loaded class is compatible with the serialized object. If no matching class is found, an InvalidClassException is thrown.

**What exactly happens when you serialize an object?**

Serialization mechanism generates serialversionuid (if you have not defined it explicitly) for the class whose object you are trying to serialize. This serialversionuid is then stored inside the file.

**What exactly happens when you deserialize an object?**

During deserialization the class whose object you want to deserialize , needs to be loaded. Here, the deserialization mechanism generates serialversionuid (if you have not defined it explicitly) for the class whose object you are trying to deserialize. This serialversionuid is then checked with the serialversionuid stored inside the file if they match then deserialization becomes successful else you get an exception "InvalidClassException" and deserialization fails.

Since the default serialVersionUID computation differs on different JVM implementations, it is highly recommended for a class which implements Serializable or Externalizable interfaces to declare serialVersionUID explicitly in order to ensure successful deserialization across all the platforms.

1. **What is a transient variable in Java?**

If you don't need to save and restore any member variable (e.g., the password kept in a String object), the private modifier will not help you. Serialized information can be read in a file or in a captured network packet. You may implement the Externalizable interface, which is demonstrated in the previous paragraph. In this case, nothing is written automatically, and you can control the entire process.

However, serializable objects are much more convenient because everything is serialized there automatically. You can forbid serialization of any member variable object with the transient modifier. It tells the JVM: "Do not save and restore this field, please; somebody else will take care of this field."

1. **When should we use transient keywords in java?**

a) First and very logical case would be where you may have fields that are derived/calculated from other fields within an instance of class. They should be calculated programmatically every time rather than having the state be persisted via serialization.

b) Second logical example can be any secure information which should not leak outside the JVM in any form (either in database OR byte stream).

c) Another example could be fields which are not marked as “Serializable” inside JDK or application code. Classes which do not implement Serializable interface and are referenced within any serializable class, cannot be serialized; and will throw a “java.io.NotSerializableException” exception. These non-serializable references should be marked “transient” before serializing the main class.

d) And lastly, there are times when it simply doesn’t make sense to serialize some fields. Period. For example, In any class if you have added a logger reference, then what's the use of serializing that logger instance. Absolutely no use.

1. **What are the things that get written when you serialize an object?**

It writes out the metadata (description) of the class associated with an instance such as length of the class, the name of the class, serialVersionUID (or serial version), the number of fields in this class.

Then it recursively writes out the metadata of the superclass until it finds java.lang.object.

Once it finishes writing the metadata information, it then starts with the actual data associated with the instance. But this time, it starts from the top most superclass.

Finally it writes the data of objects associated with the instance starting from metadata to actual content recursively.

1. **If a class is Serializable but its superclass is not, what will be the state of the instance variables inherited from superclass after deserialization?**

Java serialization process only continues in object hierarchy till the class is Serializable i.e. implements Serializable [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) in Java and values of the instance variables inherited from super class will be initialized by calling constructor of Non-Serializable Superclass during *deserialization* process.

1. **Suppose super class of a new class implements Serializable interface, how can you avoid new class from being serialized?**

If Super Class of a Class already implements Serializable interface in Java then its already Serializable in Java, since you can not un-implement an interface it's not really possible to make it Non Serializable class but yes there is a way to avoid serialization of new class. **To avoid java serialization** you need to implement writeObject() and readObject() methods in your Class and need to throw NotSerializableException from those methods. This is another benefit of *customizing java*

1. **Suppose you have a class which you serialized and stored in persistence and later modified that class to add a new field. What will happen if you deserialize the object already serialized?**

It depends on whether the class has its own SerialVersionUID or not. If we don't provide serialVersionUID in our code, the runtime system will generate it. By adding any new field there is chance that new serialVersionUID generated for that class version is not the same of already serialized object and in this case JVM will raise java.io.InvalidClassException and this is the reason *it's recommended to have your own serialVersionUID* in code and make sure to keep it same always for a single class.

1. **Which kind of variables are not serialized during Java Serialization?**

Since *static variables belong to the class* and not to an object they are not part of the state of the object so they are not saved during the Java Serialization process. Transient variables are also not included in the Java serialization process and are not part of the object’s serialized state.

1. **Does the “javalangObject” class implement “Serializable”? Explain with reason**
2. **What happens when one of the members in the class doesn't implement a Serializable interface?**

If you try to serialize an object of a class which implements Serializable, but the object includes a reference to an non- Serializable class then a ‘NotSerializableException’ will be thrown at runtime.

1. **What happens when the base class doesn't implement the Serializable interface but the subclass implements?**

If you try to serialize an object of a class which implements Serializable but its base class does not , at the most what will happen is base class members won’t get serialized. Java serialization process only continues in object hierarchy till the class is Serializable i.e. implements Serializable [interface in Java](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) and values of the instance variables inherited from super class will be initialized by calling constructor of Non-Serializable Superclass during deserialization process.

1. **Can you Customize Serialization process or can you override the default Serialization process in Java?**

The answer is yes you can. We all know that for serializing an object ObjectOutputStream.writeObject (saveThisobject) is invoked and for reading object ObjectInputStream.readObject() is invoked but there is one more thing which Java Virtual Machine provides you is to define two methods “private void writeObject(ObjectOutputStream out)” and “private void readObject(ObjectInputStream in)” in your class. If you define these two methods in your class then JVM will invoke these two methods instead of applying the default serialization mechanism. You can customize behaviour of object serialization and deserialization here by doing any kind of pre or post processing task. Important point to note is making these methods private to avoid being inherited, overridden or overloaded. Since only Java Virtual Machine can call private methods, integrity of your class will remain and Java Serialization will work as normal.

1. **While customizing the serialization process, if we define “readObject()” and “writeObject()” with accessibility modifiers other than “private” what will happen?**

If these methods are not private, they will be ignored by JVM.

1. **Can we transfer a Serialized object via network?**

Yes you can transfer a Serialized object via network because Java serialized object remains in the form of bytes which can be transmitted via network. You can also store serialized objects in Disk or database as Blob.

1. **How many ways can we create objects in java?**

**There are five total ways to create objects in Java, which are explained below.**

|  |  |
| --- | --- |
| Using new keyword | } → constructor gets called |
| Using newInstance() method of Class class | } → constructor gets called |
| Using newInstance() method of Constructor class | } → constructor gets called |
| Using clone() method | } → no constructor call |
| Using deserialization | } → no constructor call |

1. **What is the difference between Standard IO and NIO?**

First main difference between the standard IO and NIO is, standard IO is based on streams and NIO is buffer oriented. Buffer oriented operations provide flexibility in handling data. In buffer-oriented NIO, data is first read into a buffer and then it is made available for processing. So, we can move back and forth in the buffer. But in the case of streams, it is not possible. Second main difference is blocking and non-blocking IO operations. In case of streams, a thread will be blocked until it completes the IO operation. Wherein the NIO allows for non-blocking operations. If the data is not available for IO operations, then the thread can do something else and need not stay in blocked mode.

1. **What are Generics in Java?**

Generics are a facility of [generic programming](http://en.wikipedia.org/wiki/Generic_programming) that were added to the [Java programming language](http://en.wikipedia.org/wiki/Java_(programming_language)) in 2004 within [J2SE](http://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition)5.0. They allow "a type or method to operate on objects of various types while providing compile-time type safety**.**

1. **How does Generic provide Type-safety?**

The following block of Java code illustrates a problem that exists when not using generics. First, it declares an ArrayListof type Object. Then, it adds a String to the ArrayList. Finally, it attempts to retrieve the added String and cast it to an Integer.

List v = new ArrayList();

v.add("test");

Integer i = (Integer)v.get(0); // exception: ClassCastException

Although the code compiles without error, it throws a runtime exception (java.lang.ClassCastException) when executing the third line of code. This type of problem can be avoided by using generics and is the primary motivation for using generics.

Using generics, the above code fragment can be rewritten as follows:

List<String> v = new ArrayList<String>();

v.add("test");

Integer i = v.get(0); // (type error) Compile time error

The type parameter String within the angle brackets declares the ArrayList to be constituted of String. With generics, it is no longer necessary to cast the third line to any particular type, because the result of v.get(0) is defined as String by the code generated by the compiler.

Compiling the third line of this fragment with J2SE 5.0 (or later) will yield a compile-time error because the compiler will detect that v.get(0) returns String instead of Integer.

1. **What do you mean by type-erasure?**

All the generic information applied in a java application is removed by the compiler while compiling the code. This is known as “type-erasure”.

i.e. Type erasure is the technique using which the Java compiler translates generic / parameterized type to raw type in Java generics.

1. **Explain <? extends T> in java generics**

It means you can pass T or its sub type. Here you can’t add anything to the collection.

E.g.

**void disp(List<? Extends Animal> mylist)**

**{**

**}**

U can invoke disp() by passing the List implementation of Tiger. But inside disp() using mylist u can’t add anything. This is because the compiler will have no way to detect a problem caused by adding Dog inside the list which is actually meant for Tiger.

1. **Explain <? super T> in java generics**

It means you can pass T or its base type. Here you can add inside collection.

E.g.

**void disp(List<? super Dog> mylist)**

**{**

**}**

You can invoke disp() by passing say List implementation of Dog or its base type (say Animal or Object). Now inside disp() using mylist u can add Dog. This is because mylist variable will point to List implementation of Dog or Animal or Object. And you can see there is absolutely no problem in adding a Dog inside a List of Dog or Animal or Object.

1. **What is the rule of polymorphism and generics?**

Upcast applies to the "base" type of the collection and not the “generic” type.

e.g.

**Following code does not work**

**class Parent{}**

**class Child extends Parent{}**

**List<Parent>mylist=new ArrayList<Child>();**

Because the rule is, generic type of reference and generic type of the object to which it refers must be identical. Upcast applies to the "base" type, and the meaning of "base" is the collection class itself.

Here List and ArrayList are type “base” and Parent and Child are type “generics”.

1. **Given the following two scenarios:**

**a)**

**public void addAnimal(Animal arr[])**

**{**

**System.out.println(arr[0]);**

**}**

**Dog d[ ]={ new Dog(),new Dog( ) };**

**addAnimal(d);**

**b)**

**public void**

**addAnimal(List<Animal> ref)**

**{**

**System.out.println(ref);**

**}**

**addAnimal(new**

**ArrayList<Dog>());**

**Explain with reason which scenario is legal at compile-time and which is not?**

Scenario a is legal at compile-time whereas b is not.

The reason the compiler won't let u pass an ArrayList<Dog> into a method that takes an ArrayList<Animal>, is because within the method, that parameter is of type ArrayList<Animal>, and that means u could put any kind of Animal into it. There would be no way for the compiler to stop you from putting a Cat into a List that was originally declared as <Dog>, but is now referenced from the <Animal> parameter.

Now the question arises, why does the compiler allow it in case of arrays and not in collection?

The reason you can get away with compiling this for arrays is because there is a runtime exception (ArrayStoreException) that will prevent you from putting the wrong type of object into an array. If u send a Dog array into the method that takes an Animal array, and u add only Dogs (including Dogs subtypes) into the array now referenced by Animal, no problem. But if you do try to add Cat to the object that is actually a Dog array, you will get the exception.

But there is no equivalent exception for generics, because of type erasure!

In other words, at runtime the JVM knows the type of arrays but does not know the type of collection. All the generic type information is removed during compilation, so by the time it gets to the JVM, there is simply no way to recognize the disaster of putting a Cat into an ArrayList<Dog>. that is at runtime the JVM would have no way to stop u from adding a Cat to what was created as a Dog collection.

1. **What is the difference between List<?> mylist and List<Object> mylist?**

There is a huge difference. List<?>, which is the wildcard<?> without the keywords extends or super, simply means "any type". So that means any type of List can be assigned to the argument. That could be a List of <Dog>, <Integer> etc. whatever. And using the wildcard alone, without the keyword super(followed by a type),means that u cannot add anything to the list referred to as List<?>.

List<Object> is completely different from List<?>. List<Object> means that the method can take only a List<Object>. Not a List<Dog> or List<Cat>. It does, however, mean that u can add to the list, since the compiler has already made certain that u r passing only a valid List<Object> into the method.

1. **What is the importance of Collection API?**

What if you want to have a collection of data items ?

e.g. Strings, numbers, characters ,objects etc.

one way is to create an array. They are efficient.

But if we have following needs:

a) Dynamic array

b) Sorted data

c) uniqueness

d) key-value storage

e) effective way of persisting

Arrays cannot fulfill the above needs. We have to use containers for that.

Containers

Give us advantages such as

* **Dynamic**
* **Sorted Order**
* **Uniqueness**
* **Thread-safe**
* **Performance**
* **Key-value storage, convenient for retrieval by passing a key.**
* **Provide an effective way of storing and maintaining data inside the file system.**

**Iterators**

Allow us to traverse through the container.

**Algorithms**

Allow us to perform tasks such as min,max, sort etc. on the data structure.

1. **What are the basic interfaces of Java Collections Framework?**

**Collection** is the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Java platform doesn’t provide any direct implementations of this interface.

**Set** is a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards.

**List** is an ordered collection and can contain duplicate elements. You can access any element from its index. List is more like array with dynamic length.

**A Map** is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value.

Some other interfaces are **Queue, Dequeue, Iterator, SortedSet, SortedMap and ListIterator.**

1. **Why Collection doesn’t extend Cloneable and Serializable interfaces?**

Collection interface specifies a group of Objects known as elements. How the elements are maintained is left up to the concrete implementations of Collection. For example, some Collection implementations like List allow duplicate elements whereas other implementations like Set don’t. A lot of the Collection implementations have a public clone method. However, it doesn’t really make sense to include it in all implementations of Collection. This is because Collection is an abstract representation. What matters is the implementation.

The semantics and the implications of either cloning or serializing come into play when dealing with the actual implementation; so concrete implementation should decide how it should be cloned or serialized, or even if it can be cloned or serialized. So mandating cloning and serialization in all implementations is actually less flexible and more restrictive. The specific implementation should make the decision as to whether it can be cloned or serialized.

1. **Why does the Map interface doesn’t extend the Collection interface?**

Although Map interface and it’s implementations are part of Collections Framework, Map are not collections and collections are not Map. Hence it doesn’t make sense for Map to extend Collection or vice versa. If Map extends Collection interface, then where are the elements? Map contains key-value pairs and it provides methods to retrieve a list of Keys or values as Collection but it doesn’t fit into the “group of elements” pattern.

1. **What is an Iterator?**

Iterator interface provides methods to iterate over any Collection. We can get an iterator instance from a Collection using the iterator*()* method. Iterator takes the place of Enumeration in the Java Collections Framework. Iterators allow the caller to remove elements from the underlying collection during the iteration. Java Collection iterator provides a generic way for traversal through the elements of a collection and implements **Iterator Design Pattern**.

1. **What is the difference between Enumeration and Iterator interface?**

Both are used to traverse through the collection implementations. The difference is Iterators allow the caller to remove elements from the underlying collection that is not possible with Enumeration. Iterator method names have been improved to make its functionality clear.

1. **What is the difference between Iterator and ListIterator?**
2. ListIterator can be used with Lists only.
3. Iterator can traverse in forward direction only whereas ListIterator can be used to traverse in both directions.
4. ListIterator inherits from the Iterator interface and comes with extra functionalities like adding an element, replacing an element, getting index position for previous and next elements.
5. **What are different ways to iterate over a list?**

We can iterate over a list in two different ways – using iterator and using for-each loop.

List<String> strList = new ArrayList<String>();

//using for-each loop

for(String obj : strList)

{

System.out.println(obj);

}

//using iterator

Iterator<String> it = strList.iterator();

while(it.hasNext())

{

String obj = it.next();

System.out.println(obj);

}

1. **What do you understand about the iterator fail-fast property?**

Iterator fail-fast property checks for any modification in the structure of the underlying collection every time we try to get the next element. If there are any modifications found, it throws ConcurrentModificationException. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.

1. **What is the difference between iterators being fail-fast and fail-safe?**

Iterator fail-fast property checks for any modification in the structure of the underlying collection everytime we try to get the next element. If there are any modifications found, it throws ConcurrentModificationException. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.