1. **What is java program?**
2. OOPS A Java program is mostly a collection of objects talking to other objects by invoking each other's methods. Every object is of a certain type, and that type is defined by a class or an interface. Most Java programs use a collection of objects of many different types.

**Class** A class is a **blueprint** or **prototype** from which objects are created. It is like a **template** that describes the kinds of **state** and **behavior** that objects of its type support.

**Object** At runtime, when the Java Virtual Machine (JVM) **encounters the new keyword**, it will **use the appropriate class to make an object which is an instance of that class**. That object will have its **own state**, and **access to all of the behaviors** (with right visibility) defined by its class.

**State (instance variables)**  Each object (instance of a class) will have its **own unique set of instance variables as defined in the class**. Collectively, the **values assigned to an object's instance variables make up the object's state**.

**Behavior (methods)** When a programmer creates a class, creates methods for that class. **Methods are where the class' logic is stored**. Methods are where the real work gets done. They are **where algorithms get executed, and data gets manipulated**.

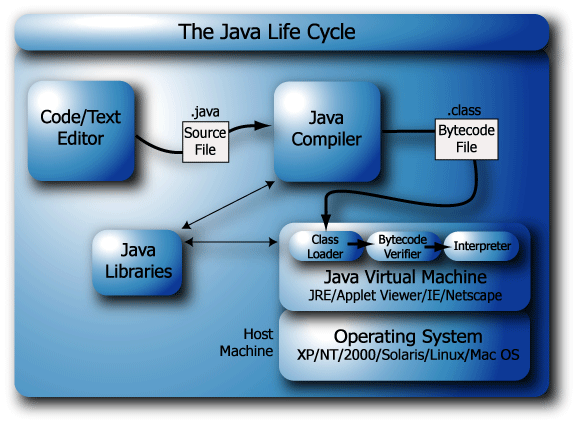
**Identifiers and Keywords** All the Java components mentioned above about—classes, variables, and methods—need names. In Java these names are called identifiers.

1. **What is JVM?**
2. When you download JRE and install on your machine you got all the code required to create JVM. Java Virtual Machine is get created when you run a java program using java command e.g. java HelloWorld. JVM is responsible for converting byte code into machine specific code and that's why you have different JVM for Windows, Linux or Solaris but one JAR can run on all this operating system. Java Virtual machine is at heart of Java programming language and provide several feature to Java programmer including Memory Management and Garbage Collection, Security and other system level services. Java Virtual Machine can be customized e.g we can specify starting memory or maximum memory of heap size located inside JVM at the time of JVM creation. If we supplied invalid argument to java command it may refuse to create Java Virtual Machine by saying "failed to create Java virtual machine: invalid argument". In short Java Virtual Machine or JVM is the one who provides Platform independence to Java.

JVM, or the Java Virtual Machine, is an interpreter which accepts ‘Bytecode’ and executes it.

Java has been termed as a ‘Platform Independent Language’ as it primarily works on the notion of ‘compile once, run everywhere’. Here’s a sequential step establishing the Platform independence feature in Java:

1. The Java Compiler outputs Non-Executable Codes called ‘Bytecode’.
2. Bytecode is a highly optimized set of computer instruction which could be executed by the Java Virtual Machine (JVM).
3. The translation into Bytecode makes a program easier to be executed across a wide range of platforms, since all we need is a JVM designed for that particular platform.
4. JVMs for various platforms might vary in configuration, those they would all understand the same set of Bytecode, thereby making the Java Program ‘Platform Independent’.



[**http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html#ixzz46EawcpTt**](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html#ixzz46EawcpTt)

1. **What is Just in Time Compiler (JIT)?**
2. Initially Java has been accused of poor performance because it’s both compiles and interpret instruction. Since compilation or Java file to class file is independent of execution of Java program do not confuse. Here compilation word is used for byte code to machine instruction translation. JIT are advanced part of Java Virtual machine which optimize byte code to machine instruction conversion part by compiling similar byte codes at same time and thus reducing overall execution time.

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "just in time" to run. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

JIT is part of Java Virtual Machine and also performs several other optimizations such as in-lining function.

http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html#ixzz46Ebc1tKA

1. **What are ClassLoaders?**

ClassLoader in Java is a class which is used to load class files in Java. Java code is compiled into class file by javac compiler and JVM executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source.

The ClassLoader is the part of the JVM.

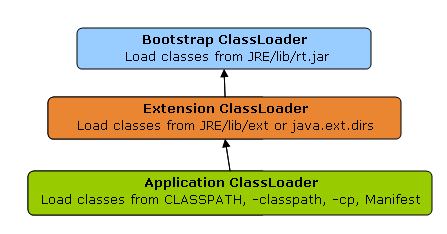
The Java ClassLoader is simply java class. This means that it's easy to create your own ClassLoader.

**ClassLoader Types –**

1. **Bootstrap -** Every class loader has a predefined location, from where they loads class files. Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar and it is parent of all class loaders in Java.

Bootstrap class loader don't have any parents. Bootstrap class loader is also known as Primordial ClassLoader in Java.

1. **Extension** - Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory or any other directory pointed by java.ext.dirs system property. Extension ClassLoader in JVM is implemented by  sun.misc.Launcher$ExtClassLoader.
2. **Application -** Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from [CLASSPATH](http://javarevisited.blogspot.sg/2011/01/how-classpath-work-in-java.html) environment variable, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class. Also, except Bootstrap class loader, which is implemented in native language mostly in C,  all  Java class loaders are implemented using java.lang.ClassLoader.



Correct understanding of class loader is must to resolve issues like NoClassDefFoundError in Java and java.lang.ClassNotFoundException, which are related to class loading.

**Classloader Principles -**

1. **Delegation** - Delegation principle tells that if class is not loaded by current classloader then it delegates class loading request to its parent classLoader so parent classLoader loads the class this is upto the top most class loader i.e Bootstrap

As discussed on when a class is loaded and initialized in Java, a class is loaded in Java, when its needed. Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in rt.jar and since that class is not there, request comes to Extension class loader which looks on jre/lib/ext directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from Classpath in Java. Remember Classpath is used to load class files while PATH is used to locate executable like javac or java command.

1. **Visibility** - Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader cannot see classes loaded by child.

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either [java.lang.ClassNotFoundException](http://javarevisited.blogspot.ca/2011/08/classnotfoundexception-in-java-example.html). as shown in below Example

**package** test;  
  
**import** java.util.logging.Level;  
**import** java.util.logging.Logger;  
  
/\*\*  
 \* Java program to demonstrate How ClassLoader works in Java,

 \* in particular about visibility principle of ClassLoader.

 \*  
 \* @author Javin Paul  
 \*/  
  
**public** **class** ClassLoaderTest {  
    
    **public** **static** **void** main(**String** args[]) {  
        **try** {            
            *//printing ClassLoader of this class*  
            **System**.out.println("ClassLoaderTest.getClass().getClassLoader() : "  
                                 + ClassLoaderTest.**class**.getClassLoader());  
  
            
            *//trying to explicitly load this class again using Extension class loader*  
            **Class**.forName("test.ClassLoaderTest", **true**   
                            ,  ClassLoaderTest.**class**.getClassLoader().getParent());  
        } **catch** (**ClassNotFoundException** ex) {  
            **Logger**.getLogger(ClassLoaderTest.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
        }  
    }  
  
}  
  
**Output:**  
ClassLoaderTest.getClass().getClassLoader() : sun.misc.Launcher$AppClassLoader@601bb1  
16/08/2012 2:43:48 AM test.ClassLoaderTest main  
SEVERE: **null**  
java.lang.**ClassNotFoundException**: test.ClassLoaderTest  
        at java.net.**URLClassLoader**$1.run(**URLClassLoader**.java:202)  
        at java.security.**AccessController**.doPrivileged(Native **Method**)  
        at java.net.**URLClassLoader**.findClass(**URLClassLoader**.java:190)  
        at sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:229)  
        at java.lang.**ClassLoader**.loadClass(**ClassLoader**.java:306)  
        at java.lang.**ClassLoader**.loadClass(**ClassLoader**.java:247)  
        at java.lang.**Class**.forName0(Native **Method**)  
        at java.lang.**Class**.forName(**Class**.java:247)  
        at test.ClassLoaderTest.main(ClassLoaderTest.java:29)

1. **Uniqueness** -Uniqueness principle allows loading a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent.

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all class loader principle while writing your own ClassLoader.

### Methods in ClassLoader -

#### loadClass

- loadClass is load the class disk into memory.but class is not initilized likely class.forName()

class MainClassLoader {  
public static void main(String[] args){  
  
ClassLoader mainLoader = MainClassLoader.class.getClassLoader();  
try {  
Class ourCLass = classLoader.loadClass("com.javawora.MainClassLoader");  
System.out.println("ourCLass.getName() = " + ourCLass.getName());  
} catch (ClassNotFoundException e) {  
e.printStackTrace();  
}  
  
}

#### defineClass -

This method mainly used for converting the raw array bytes to class object. it parses the bytecode format into a run-time data structure, checks for validity,verification etc.this method is final so we can not override it.

#### findSystemClass -

The findSystemClass method is used to loads the class files from the local filesystem.i.e bootstrap classloader after the loading the class it call defineClass method to convert raw byte array to class object   
1.classLoader requires particuler class to load  
2.check class exist or not if it is there loaded into memory.  
3.if it is not there calls findSystemClass to load from the local file system.

#### resolveClass -

if we want to load the partially then this method is used

#### findLoadedClass -

findLoadedClass works as a cache.when any classloader ask to load the class then child class loader calls findLoadedClass() so it gives all loaded class by parent classloader if it is already loaded then then it can not load the class otherwise it loads the class.This method is called first when any request come to load the class.

Now let’s get our hands dirty with some real code. Consider the following example: class A instantiates class B.

|  |
| --- |
| **public** **class** A {  **public** **void** doSomething() {  B b = **new** B();  b.doSomethingElse();  }  } |

The statement B b = new B() is semantically equivalent to

B b = A.class.getClassLoader().loadClass(“B”).newInstance()

As we see, every object in Java is associated with its class (A.class) and every class is associated with classloader (A.class.getClassLoader()) that was used to load the class.

When we instantiate a ClassLoader, we can specify a parent classloader as a constructor argument. If the parent classloader isn’t specified explicitly, the virtual machine’s system classloader will be assigned as a default parent.

<http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/lang/ClassLoader.java#ClassLoader.findLoadedClass%28java.lang.String%29>

<http://www.javawora.com/classloader>

<http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html>

<http://zeroturnaround.com/rebellabs/rebel-labs-tutorial-do-you-really-get-classloaders/>

1. **What kind of variables a class can consist of?**
2. A class consists of
3. Local variable,
4. instance variables and
5. Class variables.
6. **What is a Local Variable**
7. Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and it will be destroyed when the method has completed.

• Local (method, automatic, or stack) variable declarations cannot have access modifiers.

• final is the only modifier available to local variables.

• **Local variables don't get default values**, so they must be initialized before use

1. **What is a Instance Variable**
2. Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded.

* Instance variables can
  + - Have any access control
    - Be marked final or transient
* Instance variables can't be abstract, synchronized, native, or strictfp.
* It is legal to declare a local variable with the same name as an instance variable; **this is called "shadowing."**
* The **transient** modifier applies only to instance variables.
* The **volatile** modifier applies only to instance variables.
* A static method can't access an instance variable directly.
* **Objects and their instance variables live on the heap.**
* Instance variables live as long as their object lives.
* Instance variables are **always initialized with a default value of null**.

## The transient Modifier:

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

class Point {

int x, y;

transient float rho, theta;

}

**The volatile Modifier:**

## The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

## Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

class Test {

static volatile int i = 0, j = 0;

static void one() { i++; j++; }

static void two() {

System.out.println("i=" + i + " j=" + j);

}

}

*This allows method one and method two to be executed concurrently, but guarantees that accesses to the shared values for i and j occur exactly as many times, and in exactly the same order, as they appear to occur during execution of the program text by each thread. Therefore, the shared value for j is never greater than that for i, because each update to i must be reflected in the shared value for ibefore the update to j occurs. It is possible, however, that any given invocation of method two might observe a value for j that is much greater than the value observed for i, because method one might be executed many times between the moment when method two fetches the value of i and the moment when method two fetches the value of j.*

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.3.1.3>

<http://www.ibm.com/developerworks/library/j-5things15/>

1. **Why Java is considered dynamic?**
2. No.

**Dynamic languages** are languages that don’t necessarily need variables to be declared before they are used. Examples of dynamic languages are Python, Ruby, and PHP. So in dynamic languages the following is possible:

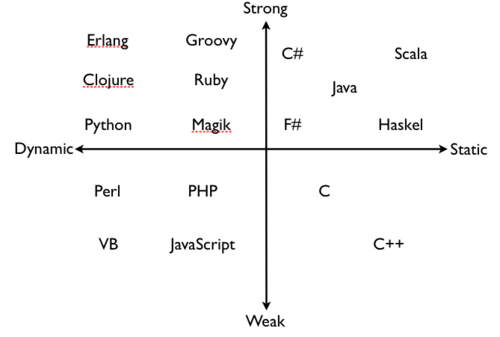
num = 10

**Static languages** are languages that variables need to be declared before use and type checking is done at compile time. Examples of static languages include Java, C, and C++. So in static languages the following is enforced

static int awesomeNumber;

awesomeNumber = 10;

<http://architects.dzone.com/articles/should-your-static-go-static>



1. **What is a Class Variable? What is Static variable?**
2. Static variables are declared with in a class, outside any method, with the static keyword.

* They are not tied to any particular instance of a class.
* No class instance is needed in order to use static members of the class.
* There is only one copy of a static variable per class and all instances share it.
* Static methods do not have direct access to non-static members.
* Non-static variables (instance variable) take on unique values with each object instance.

1. **Which class should you use to obtain design information about an object?**
2. For every type of object, the Java virtual machine instantiates an immutable instance of [java.lang.Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html" \t "_blank) which provides methods to examine the runtime properties of the object including its members and type information. [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html) also provides the ability to create new classes and objects.

**The Class class is used to obtain information about an object's design and java.lang.Class class instance represent classes, interfaces in a running Java application.**

Every time JVM creates an object, it also creates a java.lang.Class object that describes the type of the object. All instances of the same class share the same Class object and you can obtain the Class object by calling the getClass() method of the object.

By the way this method is inherited from java.lang.Object class.  
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 same class" because they are both instance of class Person.  
  
We need forName() and newInstance() because many times it happens that we don’t know the name of the class to instantiate while writing code , we may get it from config files, database, network  or from any upstream Java or C++ application.  
  
This is what we called reflective way of creating object which is one of the most powerful feature of Java and which makes way for many frameworks e.g. Spring ,Struts which uses Java reflection.  
  
Read more: <http://javarevisited.blogspot.com/2010/10/what-is-use-of-class-javalangclass-why.html#ixzz3Qz7FRClY>

1. **What is the purpose of the System class?**
2. The purpose of the System class is to provide access to system resources.

The **java.lang.System** class contains several useful class fields and methods. It cannot be instantiated. Facilities provided by System:

* standard output
* error output streams
* standard input and access to externally defined properties and environment variables.
* A utility method for quickly copying a portion of an array.
* a means of loading files and libraries

## Class declaration

Following is the declaration for **java.lang.System** class:

public final class System extends Object{

}

Following are the fields for **java.lang.System** class:

* **static PrintStream err** This is the "standard" error output stream.
* **static InputStream in** This is the "standard" input stream.
* **static PrintStream out** This is the "standard" output stream.

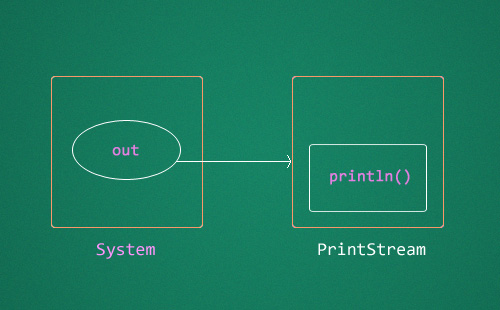
1. **How System.out.println() really works?**

[**https://luckytoilet.wordpress.com/2010/05/21/how-system-out-println-really-works/**](https://luckytoilet.wordpress.com/2010/05/21/how-system-out-println-really-works/)

[**http://javapapers.com/core-java/system-out-println/**](http://javapapers.com/core-java/system-out-println/)

**System.out.println is a Java statement that prints the argument passed, into the System.out which is generally stdout.**

* **System – is a final class** in java.lang package. As per javadoc, “…Among the facilities provided by the *System* class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an [*array*](http://javapapers.com/core-java/java-array/)…“
* **out – is a static member field of System class and is of type PrintStream**. Its [access specifiers](http://javapapers.com/core-java/access-modifiers-in-java-explain/) are public [final](http://javapapers.com/core-java/explain-the-final-keyword-in-java/). This gets instantiated during startup and gets mapped with standard output console of the host. This stream is open by itself immediately after its instantiation and ready to accept data.
* **println – is a method of PrintStream class**. println prints the argument passed to the standard console and a newline. There are multiple println methods with different arguments ([overloading](http://javapapers.com/core-java/overloading-and-overriding/)). Every println makes a call to print method and adds a newline. print calls write() and the story goes on like that.



1. **What do you mean by Constructor? What's the difference between constructors and other methods?**
2. Constructor gets invoked when a new object is created. Every class has a constructor. If we do not explicitly write a constructor for a class the java compiler builds a default constructor for that class.

* A constructor is always invoked when a new object is created.
* Each superclass in an object's inheritance tree will have a constructor called.
* Every class, even an abstract class, has at least one constructor.
* Constructors must have the same name as the class.
* **Constructors don't have a return type. If you see code with a return type, it's a method with the same name as the class, it's not a constructor.**
* Typical constructor execution occurs as follows:
  + The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the Object constructor.
  + The Object constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created.
* **Constructors can use any access modifier (even private!).** **(Use private to create singleton)**
* The compiler will create a default constructor if you don't create any constructors in your class.
* **The compiler will not create default constructor if you have created arg constructor.**
* **The default constructor is a no-arg constructor with a no-arg call to super().**
* **The first statement of every constructor must be a call to either this() (an overloaded constructor) or super().**
* The compiler will add a call to super() unless you have already put in a call to this() or super().
* Instance members are accessible only after the super constructor runs.
* Abstract classes have constructors that are called when a concrete subclass is instantiated.
* Interfaces do not have constructors.
* **If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to super() with arguments matching those of the superclass constructor.**

**public** **class** SuperOverLoadedConst {

SuperOverLoadedConst(**int** a){

System.***out***.println("a: " + a);

}

}

**public** **class** OverLoadedConst **extends** SuperOverLoadedConst {

OverLoadedConst(**int** a){

**super**(a);

System.***out***.println("a: " + a);

}

**public** **static** **void** main(String[] arg){

**new** OverLoadedConst(8);

}

}

* Constructors are never inherited, thus they cannot be overridden.
* A constructor can be directly invoked only by another constructor (using a call to super() or this()).
* Issues with calls to this()
  + May appear only as the first statement in a constructor.
  + The argument list determines which overloaded constructor is called.
  + Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call super() or the stack will explode.
  + Calls to this() and super() cannot be in the same constructor. You can have one or the other, but never both.

1. **What is the purpose of default constructor?**
2. The java compiler creates a default constructor only if there is no constructor in the class.
3. **Can a constructor be made final?**
4. No, this is not possible.
5. **Can you call one constructor from another if a class has multiple constructors?**
6. Yes, use this() syntax.
7. **List the three steps for creating an Object for a class?**
8. There are three steps when creating an object from a class:

* **Declaration:** A variable declaration with a variable name with an object type.
* **Instantiation:**The 'new' key word is used to create the object.
* **Initialization:**The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

1. **What are the datatypes in Java?**
2. The eight primitive data types supported by the Java programming language are:
3. **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large [arrays](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html), where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
4. **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
5. **int**: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -231 and a maximum value of 231-1. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 232-1. Use the Integer class to use int data type as an unsigned integer. See the section The Number Classes for more information. Static methods like compareUnsigned, divideUnsigned etc have been added to the [Integer](http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html) class to support the arithmetic operations for unsigned integers.
6. **long**: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -263 and a maximum value of 263-1. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 264-1. Use this data type when you need a range of values wider than those provided by int. The [Long](http://docs.oracle.com/javase/8/docs/api/java/lang/Long.html) class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.
7. **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3) section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the [java.math.BigDecimal](http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html" \t "_blank) class instead.[Numbers and Strings](https://docs.oracle.com/javase/tutorial/java/data/index.html) covers BigDecimal and other useful classes provided by the Java platform.
8. **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the [Floating-Point Types, Formats, and Values](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3) section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
9. **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
10. **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).
11. **String The Java programming language also provides special support for character strings via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new String object; for example, String s = "this is a string";. String objects are immutable, which means that once created, their values cannot be changed. The String class is not technically a primitive data type, but considering the special support given to it by the language**

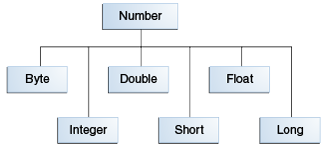
* The following chart summarizes the default values for the above data types.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Type** | **Default Value** | **Range** | **Size** | **Type** |
| byte | 0 | -128 to 127 (inclusive) | 8-bit | signed two's complement integer |
| short | 0 | -32768 to 32767 (inclusive) | 16-bit | signed two's complement integer |
| int | 0 | -2e31 to 2e31-1 (inclusive) 0 to 2e32-1(SE8 and later) | 32-bit | signed two's complement integer |
| long | 0L | -2e63 to 2e63-1 (inclusive) 0 to 2e64-1(SE8 and later) | 64-bit | signed two's complement integer |
| float | 0.0f |  | 32-bit | single-precision floating point |
| double | 0.0d |  | 64-bit | double-precision floating point |
| char | '\u0000' | \u0000' (or 0) to '\uffff' (or 65,535 inclusive) | 16-bit | single Unicode character |
| String (or any object) | null |  |  |  |
| boolean | FALSE | true and false | 1-bit |  |

1. **What are Wrapper classes?**
2. These are classes that allow primitive types to be accessed as objects. Example: Integer, Character, Double, Boolean etc.

* The wrapper classes correlate to the primitive types.
* Wrappers have two main functions:
  + To wrap primitives so that they can be handled like objects
  + To provide utility methods for primitives (usually conversions)

|  |  |  |
| --- | --- | --- |
| **Primitive** | **Wrapper Class** | **Constructor Arguments** |
| boolean | Boolean | boolean or String |
| byte | Byte | byte or String |
| char | Character | char |
| double | Double | double or String |
| float | Float | float, double, or String |
| int | Integer | int or String |
| long | Long | long or String |
| short | Short | short or String |



1. **Why do we need wrapper classes?**
2. Java uses primitive types, such as int, char, double to hold the basic data types supported by the language.

Sometimes it is required to create an object representation of these primitive types.

These are collection classes that deal only with such objects. One needs to wrap the primitive type in a class.

An example of when wrappers are used would be in Collections, you can have an **ArrayList<Integer>,** but not an **ArrayList<int>** same with HashMaps etc. **To get type safety we use generics and generics need objects not primitives.**

To satisfy this need, java provides classes that correspond to each of the primitive types. Basically, these classes encapsulate, or wrap, the primitive types within a class.

Thus, they are commonly referred to as type wrapper. Type wrappers are classes that encapsulate a primitive type within an object.

The wrapper types are Byte, Short, Integer, Long, Character, Boolean, Double, Float.

**wrapper to primitive(XXX.xxxValue())**

Primitive = wrapperObj.primitiveValue()

e.g.

Byte b = new Byte("100");

// create a byte primitive bt

byte bt = b.byteValue();

**Primitive to wrapper(XXX.valueOfxxx())**

Wrapper obj = Wrapper.valueOf(primitive)

e.g.

Integer x =Integer.valueOf(9);

Double c = Double.valueOf(5);

**String to Wrapper**

Wrapper obj = new Wrapper (String)

Wrapper obj = Wrapper.valueOf(String)

e.g.

Float a = Float.valueOf("80");

Byte b = new Byte("100");

**String to Primitive (XXX,parsexxx())**

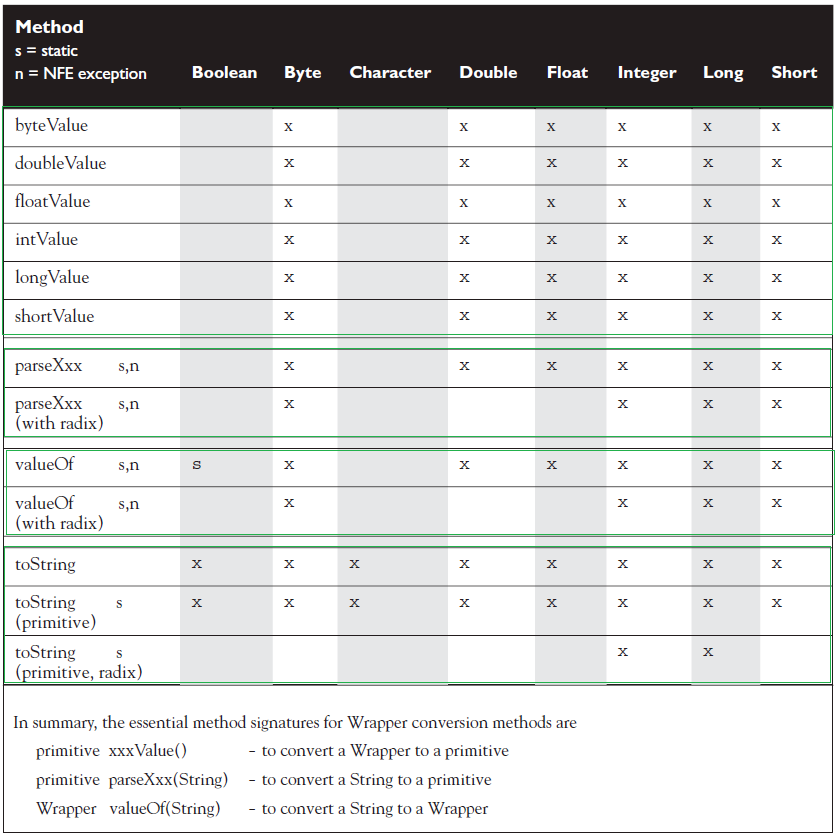
Primitive = Wrapper.parseInt(String)

**e.g.**

int x =Integer.parseInt("9");

double c = Double.parseDouble("5");

int b = Integer.parseInt("444",16);



1. **What is the difference between object oriented programming language and object based programming language?**
2. The language which itself contains objects is called as object based language and the language with follows object oriented concepts is known as object oriented language

**Object-oriented language**

Object-oriented language doesn't has in-built object. It has all feature of OOP. Object-oriented languages are C++, C#, Java etc

**Object-based language**

Object-based language doesn't support all the features of OOPs like Polymorphism and Inheritance. It has in-built object like JavaScript has window object. Languages are JavaScript, VB etc.

1. **What is static block?**
2. Static blocks are also called Static initialization blocks. A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Given below is an example:

Remember these rules:

* + A class can have any number of static initialization blocks, and they can appear anywhere in the class body.
  + The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.
  + It is used to initialize the static data member.
  + Static init blocks run once and before main method at the time of class loading by JVM.
  + JVM combines all these blocks into one single static block and then executes.
  + Instance init blocks run every time a class instance is created.
  + They run after all super-constructors and before the constructor's code has run.
  + If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.

With those rules in mind, the following output should make sense:

**public** **class** StaticBlock {

**static** {

System.*out*.println("This is first static block");

}

**public** StaticBlock(){

System.*out*.println("This is constructor");

}

**public** **static** String *staticString* = "Static Variable";

**static** {

System.*out*.println("This is second static block and "

+ *staticString*);

}

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

StaticBlock statEx = **new** StaticBlock();

StaticBlock.*staticMethod2*();

}

**static** {

*staticMethod*();

System.*out*.println("This is third static block");

}

**public** **static** **void** staticMethod() {

System.*out*.println("This is static method");

}

**public** **static** **void** staticMethod2() {

System.*out*.println("This is static method2");

}

}

Result:

This is first static block

This is second static block and Static Variable

This is static method

This is third static block

This is constructor

This is static method2

1. **Explain the following line used under Java Program:**

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

* 1. The following shows the explanation individually:
* **public**: it is the access modifier.
* **static**: it allows main() to be called without instantiating a particular instance of a class.
* **void**: it affirms the compiler that no value is returned by main().
* **main**(): this method is called at the beginning of a Java program.
* **String** **args**[ ]: args parameter is an instance array of class String

1. **What are different OOPs concepts?**
   1. The different OOps concepts are :

* [**Polymorphism**](http://java-questions.com/oops-interview-questions.html#polymorphism-java)
* [**Inheritance**](http://java-questions.com/oops-interview-questions.html#inheritance-in-java)
* [**Abstraction**](http://java-questions.com/oops-interview-questions.html#abstraction)
* [**Encapsulation**](http://java-questions.com/oops-interview-questions.html#encapsulation)
* [**Aggreagation**](http://java-questions.com/oops-interview-questions.html#aggregation)
* [**Composition**](http://java-questions.com/oops-interview-questions.html#composition)
* [**Association**](http://java-questions.com/oops-interview-questions.html#association)

1. **Define composition?**
   1. Composition is the design technique to implement has-a relationship in classes. We can use java inheritance or Object composition for code reuse.

## Association

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

http://javapapers.com/wp-content/uploads/2010/06/association.jpg

Example:A Student and a Faculty are having an association.

## Aggregation

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/aggregation.jpg

## Composition

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

http://javapapers.com/wp-content/uploads/2010/06/composition.jpg

Example:A class contains students. A student cannot exist without a class. There exists composition between class and students.

Java composition is achieved by using instance variables that refers to other objects. For example, a Person has a Job. Let’s see this with a simple code.

public class Person {

    //composition has-a relationship

    private Job job;

    public Person(){

        this.job=new Job();

        job.setSalary(1000L);

    }

    public long getSalary() {

        return job.getSalary();

    }

}

<http://www.journaldev.com/1325/what-is-composition-in-java-java-composition-example>

<http://www.journaldev.com/1775/multiple-inheritance-in-java-and-composition-vs-inheritance>

<http://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/>

1. **What is Polymorphism?**
2. Polymorphism means "many forms." Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Let us look at an example.

public interface Vegetarian{}

public class Animal{}

public class Deer extends Animal implements Vegetarian{}

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

* A Deer IS-A Animal
* A Deer IS-A Vegetarian
* A Deer IS-A Deer
* A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

Deer d = new Deer();

Animal a = d;

Vegetarian v = d;

Object o = d;

All the reference variables d,a,v,o refer to the same Deer object in the heap.

* A reference variable is always of a single, unchangeable type, but it can refer to a subtype object.
* A single object can be referred to by reference variables of many different types —as long as they are the same type or a **supertype** of the object.
* **The reference variable's type (not the object's type), determines which methods can be called!**
* **Polymorphic method invocations apply only to overridden instance methods.**

**package** com.ridd;

**public** **class** ClassParent {

**public** **void** name(){

System.*out*.println("ClassParent");

}

**public** **void** age(){

System.*out*.println("Age: " + 57);

}

**public** **void** dream (){

System.*out*.println("I am dreaming to fly");

}

}

**package** com.ridd;

**public** **class** ClassChild **extends** ClassParent{

**public** **void** name(){

System.*out*.println("ClassChild");

}

**public** **void** age(){

System.*out*.println("Age: " + 17);

}

**public** **void** fly (){

System.*out*.println("I am flying");

}

}

**package** com.ridd;

**public** **class** ClassContracter {

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

ClassParent cpcp = **new** ClassParent();

ClassParent cpcc = **new** ClassChild();

System.*out*.println(cpcp **instanceof** ClassChild); // false

**if** (cpcp **instanceof** ClassChild) {

ClassChild cccp = (ClassChild) **new** ClassParent();

}

ClassChild cccc = **new** ClassChild();

cpcp.age();

cpcc.age();

// cccp.age(); //not legal

cccc.age();

cpcp.name();

cpcc.name();

// cccp.name(); //not legal

cccc.name();

// cpcp.fly(); //not legal

// cpcc.fly(); //not legal

// cccp.name(); //not legal

cccc.fly();

cpcp.dream();

cpcc.dream();

cccc.dream();

}

}

**Result:**

false

Age: 57

Age: 17

Age: 17

ClassParent

ClassChild

ClassChild

I am flying

I am dreaming to fly

I am dreaming to fly

I am dreaming to fly

1. **What is function overloading?**
2. If a class has multiple functions by same name but different parameters, it is known as Method Overloading.

# Overloading means reusing a method name, but with different arguments.

# Overloaded methods

# Must have different argument lists

# May have different return types, if argument lists are also different

# May have different access modifiers

# May throw different exceptions

# Methods from a superclass can be overloaded in a subclass.

# Dynamic Polymorphism applies to overriding, not to overloading.

# Reference type determines which overloaded method will be used at compile time.

1. **What is function/method overriding?**
2. If a subclass provides a specific implementation of a method that is already provided by its parent class, it is known as Method Overriding.

# Methods can be overridden or overloaded; constructors can be overloaded but not overridden.

# Abstract methods must be overridden by the first concrete (non-abstract) subclass.

# With respect to the method it overrides, the overriding method

# Must have the same argument list.

# Must have the same return type, except that as of Java 5, the return type can be a subclass—this is known as a covariant return.

# Must not have a more restrictive access modifier.

# May have a less restrictive access modifier. Visibility 🡪 increase

# Must not throw new or broader checked exceptions. exception 🡪 decrease

# May throw fewer or narrower checked exceptions, or any unchecked exception.

# Final methods cannot be overridden.

# Only inherited methods may be overridden, and remember that private methods are not inherited.

# A subclass uses super.overriddenMethodName() to call the superclass version of an overridden method.

# Object type (not the reference variable's type), determines which overridden method is used at runtime.

1. **Difference between Overloading and Overriding?**

**A:**  Method overloading increases the readability of the program. Method overriding provides the specific implementation of the method that is already provided by its super class parameter must be different in case of overloading, parameter must be same in case of overriding.

|  |  |  |
| --- | --- | --- |
| **Property** | **Overloading** | **Overriding** |
| **Means** | Give more options | Replacing |
| **Method Names** | must be Same | must be Same |
| **Arg Types** | must be Different(atleast arg) | must be same(Including Order) |
| **Method Signature** | must be Different(atleast arg) | must be same(Including Order) |
| **Return Type** | No restriction | same return type or covariant |
| **Static** | No restriction | Must be Same |
| **Access Modifier** | No restriction | Same or less restrictive access modifier |
| **try/Catch** | No restriction | Exception Is thrown |
| **Method Resolution** | Compiler time (Reference) | (JVM) Run Time Polymorphism |

1. **What is runtime polymorphism or dynamic method dispatch?**
2. There are two types of polymorphism in java- Runtime polymorphism (Dynamic polymorphism) and Compile time polymorphism (static polymorphism).

[**Method overriding**](http://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) is a perfect example of runtime polymorphism. In this kind of polymorphism, reference of class X can hold object of class X or an object of any sub classes of class X. For e.g. if class Y extends class X then both of the following statements are valid:

Y obj = new Y();

//Parent class reference can be assigned to child object

X obj = new Y();

Since in method overriding both the classes (base class and child class) have same method, compile doesn’t figure out which method to call at compile-time. In this case JVM(java virtual machine) decides which method to call at runtime that’s why it is known as runtime or dynamic polymorphism.

[**Method overloading**](http://beginnersbook.com/2013/05/method-overloading/)Compile time polymorphism is nothing but the method overloading in java. In simple terms we can say that a class can have more than one method with same name but with different number of arguments or different types of arguments or both.

class X

{

void methodA(int num)

{

System.out.println ("methodA:" + num);

}

void methodA(int num1, int num2)

{

System.out.println ("methodA:" + num1 + "," + num2);

}

double methodA(double num) {

System.out.println("methodA:" + num);

return num;

}

}

class Y

{

public static void main (String args [])

{

X Obj = new X();

double result;

Obj.methodA(20);

Obj.methodA(20, 30);

result = Obj.methodA(5.5);

System.out.println("Answer is:" + result);

}

}

**Output:**

methodA:20

methodA:20,30

methodA:5.5

Answer is:5.5

As you can see in the above example that the class has three variance of methodA or we can say methodA is polymorphic in nature since it is having three different forms. In such scenario, compiler is able to figure out the method call at compile-time that’s the reason it is known as compile time polymorphism.

1. **What is final class?**

**A:**  Final classes are created so that methods implemented by that class cannot be overridden. It can’t be inherited.

1. **What do you mean by Access Modifier?**
2. Java provides access modifiers to set access levels for classes, variables, methods and constructors. A member has package or default accessibility when no accessibility modifier is specified.

**Class Access Modifiers**

* There are three access modifiers: public, protected, and private.
* There are four access levels: public, protected, **default**, and private.
* **Classes can have only public or default access.**
* A class with default access can be seen only by classes within the same package.
* A class with public access can be seen by all classes from all packages.
* Class visibility revolves around whether code in one class can
  + Create an instance of another class
  + Extend (or subclass), another class
  + Access methods and variables of another class

**Member Access Modifiers**

* Methods and instance (nonlocal) variables are known as "members."
* Members can use all four access levels: public, protected, default, private.
* Member access comes in two forms:
  + Code in one class can access a member of another class.
  + A subclass can inherit a member of its superclass.
* If a class cannot be accessed, its members cannot be accessed.
* Determine class visibility before determining member visibility.
* public members can be accessed by all other classes, even in other packages.
* If a superclass member is public, the subclass inherits it—regardless of package.
* **Members accessed without the dot operator (.) must belong to the same class.**
* this. always refers to the currently executing object.
* this.aMethod() is the same as just invoking
* **private members are not visible to subclasses, so private members cannot be inherited.**
* **Default and protected members differ only when subclasses are involved:**
  + **Default members** can be accessed only by **classes in the same package**.
  + Protected members can be accessed by other classes in the same package, plus subclasses regardless of package.
  + **protected = package plus kids** (kids meaning subclasses).
  + For subclasses outside the package, the protected member can be accessed only through inheritance; a subclass outside the package cannot access a protected member by using a reference to a superclass instance (in other words, inheritance is the only mechanism for a subclass outside the package to access a protected member of its superclass).
  + A protected member inherited by a subclass from another package is not accessible to any other class in the subclass package, except for the subclass' own subclasses.

1. **Define Inheritance?**
2. Inheritance is a mechanism wherein a new class is derived(acquires the properties) from an existing class. When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself. A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). The class from which the subclass is derived is called a superclass (also a base class or a parent class).

* Inheritance allows a class to be a subclass of a superclass, and thereby inherit public and protected variables and methods of the superclass.
* Inheritance is a key concept that underlies IS-A, polymorphism, overriding, overloading, and casting.
* All classes (except class Object), are subclasses of type Object, and therefore they inherit Object's methods.

<http://www.artima.com/lejava/articles/designprinciples.html>

1. **Is there any limitation of using Inheritance?**
2. Yes, since inheritance inherits everything from the super class and interface, it may make the subclass too clustering and sometimes error-prone when dynamic overriding or dynamic overloading in some situation.

* One of the main disadvantages of inheritance in Java (the same in other object-oriented languages) is the **increased time/effort it takes the program to jump through all the levels of overridden classes**. If a given class has ten levels of abstraction above it, then it will essentially take ten jumps to run through a function defined in each of those classes.
* Main disadvantage of using inheritance is that the two classes (base and inherited class) get **tightly coupled**. This means one cannot be used independent of each other.
* Also with time, during maintenance adding new features both base as well as derived classes are required to be changed. If a method signature is changed then we will be affected in both cases (inheritance & composition)
* If a method is deleted in the "super class" or aggregate, then we will have to re-factor in case of using that method. Here things can get a bit complicated in case of inheritance because our programs will still compile, but the methods of the subclass will no longer be overriding superclass methods. These methods will become independent methods in their own right.

<http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html>

1. **Explain the use of sublass in a Java program?**

A.  Sub class inherits all the public and protected methods and the implementation. It also inherits all the default modifier methods and their implementation.

1. **When super keyword is used?**
2. If the method overrides one of its superclass's methods, overridden method can be invoked through the use of the keyword super. It can be also used to refer to a hidden field

* **The default constructor is a no-arg constructor with a no-arg call to super().**
* **The first statement of every constructor must be a call to either this() (an overloaded constructor) or super().**
* The compiler will add a call to super() unless you have already put in a call to this() or super().
* Instance members are accessible only after the super constructor runs.
* **If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to super() with arguments matching those of the superclass constructor.**

1. **Explain the usage of this() with constructors?**
2. Within an instance method or a constructor, this is a reference to the current object — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

* Issues with calls to this()
  + May appear only as the first statement in a constructor.
  + The argument list determines which overloaded constructor is called.
  + Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call super() or the stack will explode.
  + Calls to this() and super() cannot be in the same constructor. You can have one or the other, but never both.

public class Rectangle {

private int x, y;

private int width, height;

public Rectangle() {

**this(0, 0, 1, 1);**

}

public Rectangle(int width, int height) {

**this(0, 0, width, height);**

}

public Rectangle(int x, int y, int width, int height) {

this.x = x;

this.y = y;

this.width = width;

this.height = height;

}

...

}

1. **What is Abstraction?**
   * 1. In Object oriented programming Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words **user will have the information on what the object does instead of how it does it.**

In Java Abstraction are achieved using Abstract classes, and Interfaces.

In [object-oriented programming](http://searchsoa.techtarget.com/definition/object-oriented-programming), abstraction is one of three central principles (along with [encapsulation](http://searchnetworking.techtarget.com/definition/encapsulation) and [inheritance](http://searchcio-midmarket.techtarget.com/definition/inheritance)). Through the process of abstraction, a programmer hides all but the relevant data about an [object](http://searchsoa.techtarget.com/definition/object) in order to reduce complexity and increase efficiency. In the same way that abstraction sometimes works in art, the object that remains is a representation of the original, with unwanted detail omitted. The resulting object itself can be referred to as an abstraction, meaning a [named entity](http://searchbusinessanalytics.techtarget.com/definition/named-entity) made up of selected attributes and behavior specific to a particular usage of the originating entity. **Abstraction is related to both**[**encapsulation**](http://searchnetworking.techtarget.com/definition/encapsulation)**and**[**data hiding**](http://searchsqlserver.techtarget.com/definition/data-hiding)**.**

In the process of abstraction, the programmer tries to ensure that the [entity](http://whatis.techtarget.com/definition/entity) is named in a manner that will make sense and that it will have all the relevant aspects included and none of the extraneous ones. A real-world analogy of abstraction might work like this: You (the object) are arranging to meet a blind date and are deciding what to tell them so that they can recognize you in the restaurant. You decide to include the information about where you will be located, your height, hair color, and the color of your jacket. This is all data that will help the procedure (your date finding you) work smoothly. You should include all that information. On the other hand, there are a lot of bits of information about you that aren't relevant to this situation: your social security number, your admiration for obscure films, and what you took to "show and tell" in fifth grade are all irrelevant to this particular situation because they won't help your date find you. However, since entities may have any number of abstractions, you may get to use them in another procedure in the future.

The bulk of meaning to abstraction comes from how and why it is used.

It is used for the following scenarios

1. Reduce complexity. (Create a simple interface)
2. Allow for implementation to be modified without impacting its users.
3. Create a common interface to support polymorphism (treating all implementations of the abstracted layer the same.
4. Force users to extend the implementation rather than modify.
5. Support cross platform by changing the implementation per platform.
6. **What is Abstract class**
7. An abstract class means that nobody can ever make a new instance of that class. You can still use that abstract class as a declared reference type, for the purpose of polymorphism,

An abstract class has virtually no use, no value, no purpose in life, unless it is extended.

* A class cannot be both final and abstract.
* **A final class cannot be subclassed.**
* **An abstract class cannot be instantiated.**
* **A single abstract method in a class means the whole class must be abstract.**
* An abstract class can have both abstract and nonabstract methods.
* The first concrete class to extend an abstract class must implement all of its abstract methods.
* An interface is like a 100-percent abstract class, and **is implicitly abstract whether you type the abstract modifier in the declaration or not**.
* Every class, even an abstract class, has at least one constructor.
* Abstract classes have constructors that are called when a concrete subclass is instantiated.
* **Interfaces do not have constructors.**

1. **When Abstract methods are used?**
2. 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](http://javarevisited.blogspot.com/2012/02/difference-between-linkedlist-vs.html) and [ArrayList](http://javarevisited.blogspot.com/2012/03/how-to-loop-arraylist-in-java-code.html) are free from burden of implementing all methods, had they implemented List interface directly.

j[ava.lang.Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \o "class in java.lang)

[java.util.AbstractCollection](https://docs.oracle.com/javase/7/docs/api/java/util/AbstractCollection.html)<E>

[java.util.AbstractList](https://docs.oracle.com/javase/7/docs/api/java/util/AbstractList.html)<E>

java.util.ArrayList<E>

Consider using abstract classes if any of these statements apply to your situation:

* You want to share code among several **closely related classes.**
* You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
* You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using interfaces if any of these statements apply to your situation:

* You expect that **unrelated classes** would implement your interface. For example, the interfaces [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) and [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html" \t "_blank) are implemented by many unrelated classes.
* You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
* You want to take advantage of multiple inheritance of type.

<http://www.javaworld.com/article/2077421/learn-java/abstract-classes-vs-interfaces.html>

1. **What is Encapsulation?**
2. Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason**, encapsulation is also referred to as data hiding.**

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

**Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.**

To achieve encapsulation in Java

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

1. **What is the primary benefit of Encapsulation?**
2. Encapsulation has several advantages, few of them are as shown below:

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.
* The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

1. **What is an Interface?**
2. In the Java programming language, an interface is a reference type, similar to a class that can contain only constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.

**JAVA 8 onwards we can create default and static methods with implementation inside an interface.**

<https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>

1. **Give some features of Interface?**
2. **Interface Implementation**

* Interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.
* Interfaces can be implemented by any class, from any inheritance tree.
* An interface is like a 100-percent abstract class, and **is implicitly abstract whether you type the abstract modifier in the declaration or not**.
* An interface can have only abstract methods, no concrete methods allowed.
* **JAVA 8** onwards we can create default and static methods with implementation inside an interface.
* **Interface methods are by default public and abstract**—explicit declaration of these modifiers is optional.
* **Interfaces can have constants, which are always implicitly public, static, and final**.
* Interface constant declarations of public, static, and final are optional in any combination.
* A legal nonabstract implementing class has the following properties:
  + It provides concrete implementations for the interface's methods.
  + It must follow all legal override rules for the methods it implements.
  + It must not declare any new checked exceptions for an implementation method.
  + It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.
  + It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.
  + It must maintain the exact signature (allowing for covariant returns) and return type of the methods it implements (but does not have to declare the exceptions of the interface).
* A class implementing an interface can itself be abstract.
* An abstract implementing class does not have to implement the interface methods (but the first concrete subclass must).
* A class can extend only one class (no multiple inheritance), but it can implement many interfaces.
* **An interface can extend multiple interfaces.**
* Interfaces cannot extend a class, or implement a class or interface.

1. **What is the difference between an Interface and an Abstract class?**
   * 1. Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can have static methods, main method and constructor**. | Interface **can't have static methods, main method or constructor**. |
| 5) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

<http://www.javatpoint.com/difference-between-abstract-class-and-interface>

1. **According to Java Operator precedence, which operator is considered to be with highest precedence?**
2. Postfix operators i.e () [] . is at the highest precedence.

* **Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.**
* The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first**. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.**

|  |  |
| --- | --- |
| **Operator Precedence** | |
| **Operators** | **Precedence** |
| Postfix | *expr*++ *expr*-- |
| Unary | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| multiplicative | \* / % |
| Additive | + - |
| Shift | << >> >>> |
| relational | < > <= >= instanceof |
| Equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| Ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

1. **Variables used in a switch statement can be used with which datatypes?**
2. Switch statements can evaluate only to
3. Enums,
4. byte,
5. short,
6. int, and
7. char data types.

* **You can't say,**
  + - long s = 30;
    - switch(s) { }
* **The case constant must be a literal or final variable, or a constant expression, including an enum. You cannot have a case that includes a nonfinal variable, or a range of values.**
* If the condition in a switch statement matches a case constant, execution will run through all code in the switch following the matching case statement until a break statement or the end of the switch statement is encountered. In other words, the matching case is just the entry point into the case block, but unless there's a break statement, the matching case is not the only case code that runs.
* The default keyword should be used in a switch statement if you want to run some code when none of the case values match the conditional value.
* **The default block can be located anywhere in the switch block, so if no case matches, the default block will be entered, and if the default does not contain a break, then code will continue to execute (fall-through) to the end of the switch or until the break statement is encountered.**

1. **How we can convert string to a numeric?**
2. This method is used to get the primitive data type of a certain String.

double d4 = Double.parseDouble("3.14"); //convert a String

// to a primitive

System.out.println("d4 = " + d4); // result is d4 = 3.14

There are two ways to convert string to numeric. Notice Highlighted Types.

Integer x = Integer.valueOf(str);

// or

int y = Integer.parseInt(str);

1. **Define immutable object?**
2. **An immutable object can’t be changed once it is created.**

**An object is considered immutable if its state cannot change after it is constructed.** Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption

The following rules define a simple strategy for creating immutable objects.

* 1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
  2. Make all fields final and private.
  3. Don't allow subclasses to override methods.

The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.

* 1. If the instance fields include references to mutable objects, don't allow those objects to be changed:
  + Don't provide methods that modify the mutable objects.
  + Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

**"Classes should be immutable unless there's a very good reason to make them mutable....If a class cannot be made immutable, limit its mutability as much as possible."**

1. **Why String class is considered immutable?**
2. The String class is immutable; so that once it is created a String object cannot be changed. Since String is immutable it can safely be shared between many threads, which are considered very important for multithreaded programming.

**Case 1:**

String x = "Java";

x.concat(" Rules!");

System.out.println("x = " + x); // the output is: x = Java

The first line is straightforward: create a new String object, give it the value "Java", and refer x to it. Next the VM creates a second String object with the value "Java Rules!" but nothing refers to it. The second String object is instantly lost; you can't get to it. The reference variable x still refers to the original String with the value "Java".

**Case 2:**

String x = "Java";

x = x.concat(" Rules!"); // Now we're assigning the

// new String to x

System.out.println("x = " + x); // the output will be:

// x = Java Rules!

This time, when the VM runs the second line, a new String object is created with the value of "Java Rules!", and x is set to reference it. But wait, there's more—now the original String object, "Java", has been lost, and no one is referring to it. So in both examples we created two String objects and only one reference variable, so one of the two String objects was left out in the cold.

**To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.)** Now we can start to see why making String objects immutable is such a good idea. If several reference variables refer to the same String without even knowing it, it would be very bad if any of them could change the String's value. You might say, "Well that's all well and good, but what if someone overrides the String class functionality; couldn't that cause problems in the pool?" That's one of the main reasons that the String class is marked final. Nobody can override the behaviours of any of the String methods, so you can rest assured that the String objects you are counting on to be immutable will, in fact, be immutable.

Let's look at a couple of examples of how a String might be created, and let's further assume that no other String objects exist in the pool:

String s = "abc"; // creates one String object and one

// reference variable

In this simple case, "abc" will go in the pool and s will refer to it.

String s = new String("abc"); // creates two objects,

// and one reference variable

In this case, because we used the new keyword, Java will create a new String object in normal (nonpool) memory, and s will refer to it. In addition, the literal "abc" will be placed in the pool.

* String objects are immutable, and String reference variables are not.
* If you create a new String without assigning it, it will be lost to your program.
* If you redirect a String reference to a new String, the old String can be lost.
* String methods use zero-based indexes, except for the second argument of substring().
* The String class is final—its methods can't be overridden.
* When the JVM finds a String literal, it is added to the String literal pool.
* Strings have a method: length(); arrays have an attribute named length.

String objects are immutable, so if you choose to do a lot of manipulations with String objects, you will end up with a lot of abandoned String objects in the String pool. (Even in these days of gigabytes of RAM, it's not a good idea to waste precious memory on discarded String pool objects.) On the other hand, objects of type StringBuffer and StringBuilder can be modified over and over again without leaving behind a great effluence of discarded String objects.

1. **List String class methods.**

<http://www.tutorialspoint.com/java/java_strings.htm>

* **charAt()** Returns the character located at the specified index

**Code:**

String s = "Strings are immutable";

char result = s.charAt(8);

**Output:**

a

* **concat()** Appends one String to the end of another ( "+" also works)
* **equalsIgnoreCase()** Determines the equality of two Strings, ignoring case
* **length()** Returns the number of characters in a String
* **replace()** Replaces occurrences of a character with a new character
* **substring()** Returns a part of a String
* **toLowerCase()** Returns a String with uppercase characters converted
* **toString()** Returns the value of a String
* **toUpperCase()** Returns a String with lowercase characters converted
* **trim()** Removes whitespace from the ends of a String
* **indexOf()**

**variant:**

* public int indexOf(int ch )
* public int indexOf(int ch, int fromIndex)
* int indexOf(String str)
* int indexOf(String str, int fromIndex)

public static void main(String args[]) {

String Str = new String("Welcome to Tutorialspoint.com");

String SubStr1 = new String("Tutorials");

String SubStr2 = new String("Sutorials");

System.out.print("Found Index :" );

System.out.println(Str.indexOf( 'o' ));

System.out.print("Found Index :" );

System.out.println(Str.indexOf( 'o', 5 ));

System.out.print("Found Index :" );

System.out.println( Str.indexOf( SubStr1 ));

System.out.print("Found Index :" );

System.out.println( Str.indexOf( SubStr1, 15 ));

System.out.print("Found Index :" );

System.out.println(Str.indexOf( SubStr2 ));

}

**Output:**

Found Index :4

Found Index :9

Found Index :11

Found Index :-1

Found Index :-1

1. **Why StringBuffer is called mutable?**
2. The String class is considered as immutable; so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters then StringBuffer should be used.

**StringBuffer**

StringBuffer sb = new StringBuffer("abc");

sb.append("def");

System.out.println("sb = " + sb); // output is "sb = abcdef"

1. **What is the difference between StringBuffer and StringBuilder class?**
2. StringBuilder

StringBuilder sb = new StringBuilder("abc");

sb.append("def").reverse().insert(3, "---");

System.out.println( sb ); // output is "fed---cba"

* The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.
* StringBuilder methods should run faster than StringBuffer methods.

1. **Which package is used for pattern matching with regular expressions?**
2. A: java.util.regex package is used for this purpose.
3. **java.util.regex consists of which classes?**
4. The java.util.regex package primarily consists of the following three classes:

<http://tutorials.jenkov.com/java-regex/pattern.html>

* **Pattern Class:** The Java Pattern class (java.util.regex.Pattern), is the main access point of the Java regular expression API. Whenever you need to work with regular expressions in Java, you start with Java's Pattern class.

The Java Pattern class can be used in two ways. You can use the Pattern.matches() method to quickly check if a text (String) matches a given regular expression. Or you can compile a Pattern instance using Pattern.compile() which can be used multiple times to match the regular expression against multiple texts.

* **Matcher Class:** Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the matcher method on a Pattern object.

The Java Matcher class (java.util.regex.Matcher) is used to search through a text for multiple occurrences of a regular expression. You can also use a Matcher to search for the same regular expression in different texts.

* **PatternSyntaxException:** A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Example:

import java.util.regex.Pattern;

public class PatternMatchesExample {

public static void main(String[] args) {

String text =

"This is the text to be searched " +

"for occurrences of the pattern.";

String pattern = ".\*is.\*";

boolean matches = Pattern.matches(pattern, text);

System.out.println("matches = " + matches);

}

}

**import** java.util.regex.Pattern;

**import** java.util.regex.Matcher;

**public** **class** MatcherExample {

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

String text = "This is the text to be searched "

+ "for occurrences of the pattern.";

String patternString = "This";

Pattern pattern = Pattern.*compile*(patternString);

Matcher matcher = pattern.matcher(text);

**boolean** matches = matcher.matches();

System.***out***.println("lookingAt = " + matcher.lookingAt());

System.***out***.println("matches = " + matches);

}

}

**Result:**

lookingAt = true

matches = false

This example matches the regular expression "this is the" against both the beginning of the text, and against the whole text. Matching the regular expression against the beginning of the text (lookingAt()) will return true.

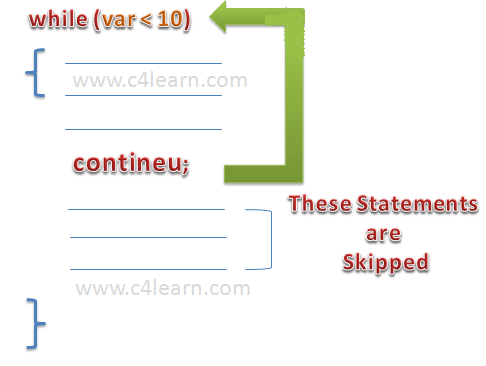
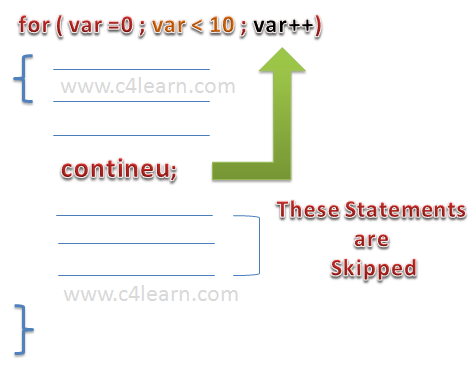
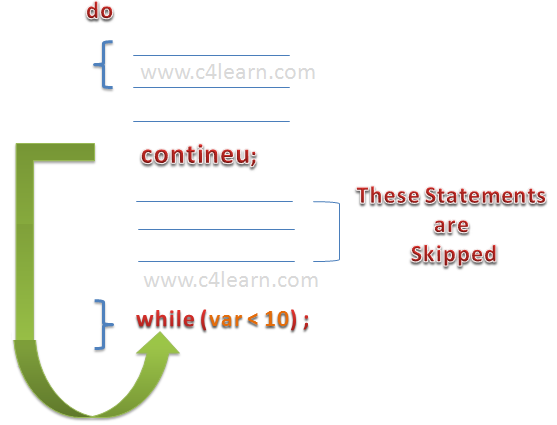
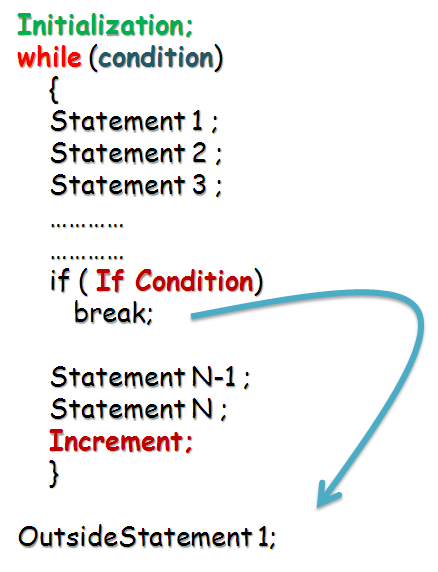
1. **Which Java operator is right associative?**
   * 1. The = operator is right associative.

**Associativity**. When an expression has two operators with the same precedence, the expression is evaluated according to its associativity. For example x = y = z = 17 is treated as x = (y = (z = 17)), leaving all three variables with the value 17, since the = operator has right-to-left associativity (and an assignment statement evaluates to the value on the right hand side). On the other hand, 72 / 2 / 3 is treated as (72 / 2) / 3 since the / operator has left-to-right associativity.

1. **What is the difference between a break statement and a continue statement?**
2. A break statement results in the termination of the statement to which it applies (switch, for, do, or while). A continue statement is used to end the current loop iteration and return control to the loop statement.

**Using break and continue**

* An unlabeled break statement will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.
* An unlabeled continue statement will cause: the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.
* If the break statement or the continue statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

****

1. **Break statement can be used as labels in Java?**
2. Yes, an example can be *break one;*

**package** ridd.examples;

**public** **class** BreakLabel {

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

**int**[][] intArray = **new** **int**[][] { { 1, 2, 3, 4, 5 },

{ 10, 20, 30, 40, 50 } };

**boolean** blnFound = **false**;

System.*out*.println("Searching 30 in two dimensional int array..");

Outer: **for** (**int** intOuter = 0; intOuter < intArray.length; intOuter++) {

System.*out*.println("Outer loop run: " + intOuter );

Inner: **for** (**int** intInner = 0; intInner < intArray[intOuter].length; intInner++) {

System.*out*.println("Inner loop run: " + intInner );

**if** (intArray[intOuter][intInner] == 30) {

blnFound = **true**;

**break** Outer;

}

}

}

**if** (blnFound == **true**)

System.*out*.println("30 found in the array");

**else**

System.*out*.println("30 not found in the array");

}

}

Result:

Searching 30 in two dimensional int array..

Outer loop run: 0

Inner loop run: 0

Inner loop run: 1

Inner loop run: 2

Inner loop run: 3

Inner loop run: 4

Outer loop run: 1

Inner loop run: 0

Inner loop run: 1

Inner loop run: 2

30 found in the array

1. **What will happen if static modifier is removed from the signature of the main method?**
2. Program throws "NoSuchMethodError" error at runtime .
3. **What is the default value of an object reference declared as an instance variable?**
4. Null, unless it is defined explicitly.
5. **Can a top level class be private or protected?**
6. No, a top level class cannot be private or protected.

Classes can have only public or default/no modifier access.

1. **What is the Locale class?**
2. public final class Locale

extends Object

implements Cloneable, Serializable

**A Locale object represents a specific geographical, political, or cultural region.** An operation that requires a Locale to perform its task is called locale-sensitive and uses the Locale to tailor information for the user. For example, displaying a number is a locale-sensitive operation— the number should be formatted according to the customs and conventions of the user's native country, region, or culture.

1. **What is dot operator?**
2. The dot operator(.) is used to access the instance variables and methods of class objects. It is also used to access classes and sub-packages from a package.
3. **Where and how can you use a private constructor?**
4. Private constructor is used if you do not want other classes to instantiate the object (Singleton Class) and to prevent sub classing.
5. **What is Singleton class?**
6. Singleton class controls object creation, limiting the number to one but allowing the flexibility to create more objects if the situation changes.

public class SingleObject {

//create an object of SingleObject

private static volatile Singleton \_instance; //volatile variable

//make the constructor private so that this class cannot be instantiated

private SingleObject(){}

//Get the only object available

public static SingleObject getInstance(){

if(\_instance == null){

synchronized(Singleton.class){

if(\_instance == null)

\_instance = new Singleton();

}

}

return instance;

}

}

public class MainClass {

static SingletonClass single;

public static void main(String[] args) {

// TODO Auto-generated method stub

single = SingletonClass.getInstance();

single.showMessage();

}

}

**Read more:[http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html#ixzz3v41OaDgj](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html" \l "ixzz3v41OaDgj)**

1. **What is type casting?**
2. Type casting means treating a variable of one type as though it is another type.

There can be 2 casting java scenarios

· **Upcasting:** Up-casting is casting to a supertype. This is Implicit casting (widening conversion).

Dog dog = new Dog();

Animal animal = (Animal) dog;

· **Downcasting**: Downcasting is casting to a subtype. Subcasting involves a type check and can throw a ClassCastException.

Animal animal = new Dog();

Dog castedDog = (Dog) animal;

Below code will throw ClassCastException.

Animal a = new Animal();

Dog d = (Dog) a;

1. **What is the difference between the >> and >>> operators?**

**A:** The >> operator carries the sign bit when shifting right. The >>> zero-fills bits that have been shifted out.

<< signed left shift - shifts a bit pattern to the left

0 0 1 1 1 => 0 1 1 1 0

>> signed right shift - shifts a bit pattern to the right

0 0 1 1 1 => 0 0 0 1 1

>>> unsigned right shift - shifts a zero into the leftmost position

1 1 1 0 => 0 0 1 1

~ unary bitwise complement operator

A | Result

0 | 1

1 | 0

0 | 1

1 | 0

& bitwise and

A | B | Result

0 | 0 | 0

1 | 0 | 0

0 | 1 | 0

1 | 1 | 1

^ xor

A | B | Result

0 | 0 | 0

1 | 0 | 1

0 | 1 | 1

1 | 1 | 0

| inclusive or

A | B | Result

0 | 0 | 0

1 | 0 | 1

0 | 1 | 1

1 | 1 | 1

1. **Does Java allow Default Arguments?**

**A:** No, Java does not allow Default Arguments.

1. **Which number is denoted by leading zero in java?**

**A:** Octal Numbers are denoted by leading zero in java, example: 06

#### Decimal to octal conversion examples

* (24)10 = (30)8
* (112)10 = (160)8
* (2048)10 = (4000)8

##### **Decimal Octal Conversion Chart Table**

|  |  |
| --- | --- |
| **Decimal** | **Octal** |
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 10 |
| 9 | 11 |
| 10 | 12 |
| 11 | 13 |
| 12 | 14 |
| 13 | 15 |
| 14 | 16 |
| 15 | 17 |
| 16 | 20 |

String

1. **What is ENUM?**

Enum in java is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc.

The java enum constants are static and final implicitly. It is available from JDK 1.5.

Java Enums can be thought of as classes that have fixed set of constants.

* enum improves type safety
* enum can be easily used in switch
* enum can be traversed
* enum can have fields, constructors and methods
* enum may implement many interfaces but cannot extend any class because it internally extends Enum class

**valueOf()**

The **java.lang.Enum.valueOf()** method returns the enum constant of the specified **enumtype** with the specified name. The **name** must match exactly an identifier used to declare an enum constant in this type.

**values()**

The java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

class EnumExample1{

public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args) {

for (Season s : Season.values())

System.out.println(s);

}}

Internal code generated by the compiler for the above example of enum type

The java compiler internally creates a static and final class that extends the Enum class as shown in the below example:

public static final class EnumExample1$Season extends Enum

{

private EnumExample1$Season(String s, int i)

{

super(s, i);

}

public static EnumExample1$Season[] values()

{

return (EnumExample1$Season[])$VALUES.clone();

}

public static EnumExample1$Season valueOf(String s)

{

return (EnumExample1$Season)Enum.valueOf(EnumExample1$Season, s);

}

public static final EnumExample1$Season WINTER;

public static final EnumExample1$Season SPRING;

public static final EnumExample1$Season SUMMER;

public static final EnumExample1$Season FALL;

private static final EnumExample1$Season $VALUES[];

static

{

WINTER = new EnumExample1$Season("WINTER", 0);

SPRING = new EnumExample1$Season("SPRING", 1);

SUMMER = new EnumExample1$Season("SUMMER", 2);

FALL = new EnumExample1$Season("FALL", 3);

$VALUES = (new EnumExample1$Season[] {

WINTER, SPRING, SUMMER, FALL

});

}

}

<http://howtodoinjava.com/java-5/guide-for-understanding-enum-in-java/>

<http://www.javatpoint.com/enum-in-java>

<http://stackoverflow.com/questions/20662018/whats-difference-between-enums-and-final-variables>

1. **What is static class?**

https://www.caveofprogramming.com/java/java-static-class-tutorial.html