1. **What do you know about Java?**
2. Java is
3. Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible.
4. It is intended to let application developers "write once, run anywhere" (WORA) meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.
5. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture
6. A high-level programming language
7. Originally Developed by Sun Microsystems and released in 1995.
8. Runs on a variety of platforms,

* Such as Windows,
* Mac OS, and
* The various versions of UNIX

Refer core JAVA doc.

1. **What is Java program?**
2. 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 are the supported platforms by Java Programming Language?**
2. Java runs on a variety of platforms, such as
3. Windows,
4. Mac OS, and
5. the various versions of UNIX/Linux like HP-Unix,
6. Sun Solaris,
7. Redhat Linux,
8. Ubuntu,
9. Android
10. CentOS, etc.
11. **List any five features of Java?**
12. Some features include
13. Object Oriented
14. Platform Independent (WORA)
15. Robust
16. Interpreted
17. Multi-threaded
18. **Why is Java Architectural Neutral?**

**A:**  A Java program has to be converted to a form which Java VM can understand so any computer with a Java VM can interpret and run the program.

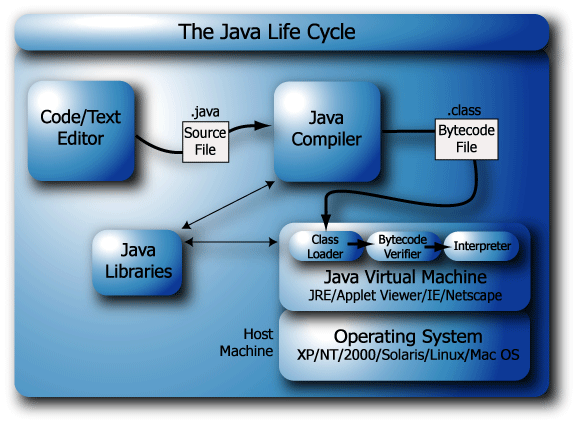
Compiling a Java program means taking the programmer-readable text in your program file (also called source code) and converting it to bytecodes, which are platform-independent instructions for the Java VM.

Once your program successfully compiles into Java bytecodes, you can interpret and run applications on any Java VM. Interpreting and running a Java program means invoking the **Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes(binary)** so your computer can understand and run the program.

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’.



1. **How Java enabled High Performance? What is JIT compiler?**
2. Java uses Just-In-Time compiler to enable high performance.

<http://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm>

The Just-In-Time (JIT) compiler is a component of the Java™ Runtime Environment that improves the performance of Java applications at run time.

Java programs consist of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run 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. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

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.

<https://www-01.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.aix.71.doc/diag/understanding/jit_overview.html>

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:

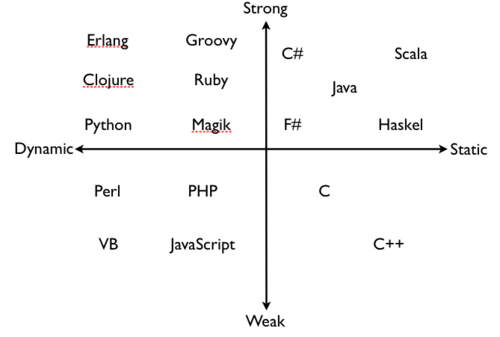
num = 10

**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. **List two Java IDE’s?**
3. Netbeans,
4. Eclipse,
5. intelliJ etc.
6. **List some Java keywords(unlike C, C++ keywords)?**
7. Some Java keywords are import, super, finally, etc.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Abstract | continue | for | new | switch |
| assert\*\*\* | default | goto\* | package | synchronized |
| Boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum\*\*\*\* | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp\*\* | volatile |
| const\* | float | native | super | while |

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. **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. **What are ClassLoaders?**
2. A class loader is an object that is responsible for loading classes. The class ClassLoader is an abstract class.

The lazy nature of Java has an effect on how do classloaders work – everything should be done at the last possible moment. A class will be loaded only when it is referenced somehow – by calling a constructor, a static method or field.

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

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**

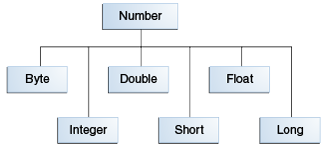
* The following chart summarizes the default values for the above data types.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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())**

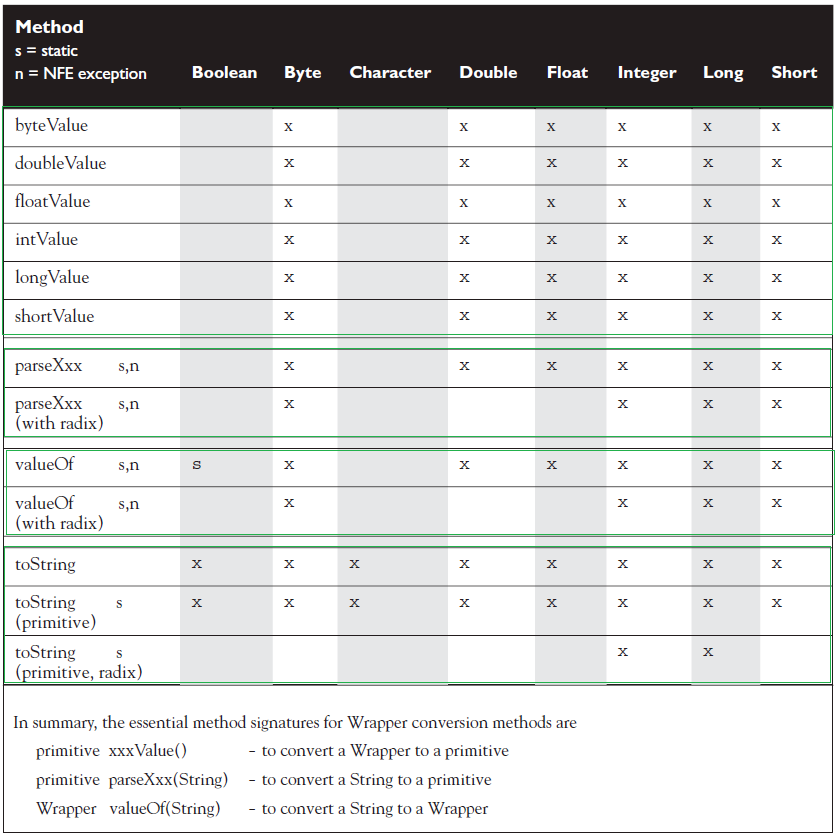
Primitive = Wrapper.parseInt(String)

**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.

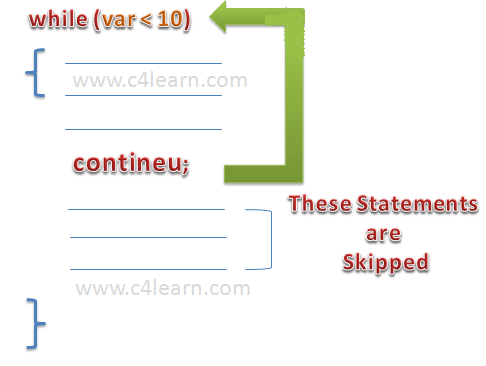
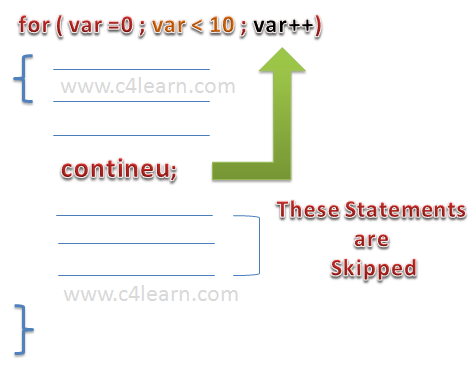
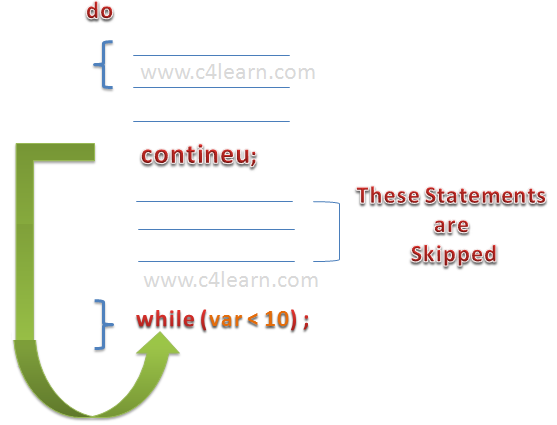
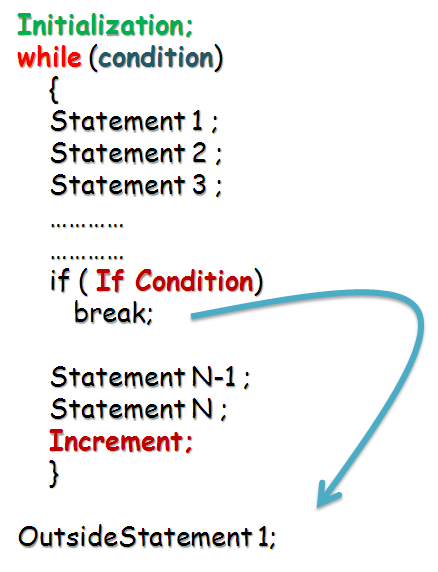
1. **Which Java operator is right associative?**
   * 1. The = operator is right associative.

**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>**

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 |

**XE:Exception**

1. **Difference between throw and throws?**
   * 1. It includes:
   * Throw is used to trigger an exception where as throws is used in declaration of exception. The throws keyword appears at the end of a method's signature.
   * Without throws, Checked exception cannot be handled where as checked exception can be propagated with throws.

**Throws:**

public void myMethod() throws PRException

{This means the super function calling the function should be equipped to handle this exception.

public void Callee()

{

try{

myMethod();

}catch(PRException ex)

{

...handle Exception....

}

}

}

**Throw:**

try{

if(age>100)

{ throw new AgeBarException(); //Customized ExceptioN

}else{

....

}

}

}catch(AgeBarException ex){

...handle Exception.....

}

1. **Can try statements be nested?**
2. Yes

**class** NestTry {

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

**try** {

**int** a = args.length;

**int** b = 42 / a;

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

**try** {

**if** (a == 1)

a = a / (a - a); // division by zero

**if** (a == 2) {

**int** c[] = { 1 };

c[42] = 99; // generate an out-of-bounds exception

}

} **catch** (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index out-of-bounds: " + e);

}

} **catch** (ArithmeticException e) {

System.out.println("Divide by 0: " + e);

}

}

}

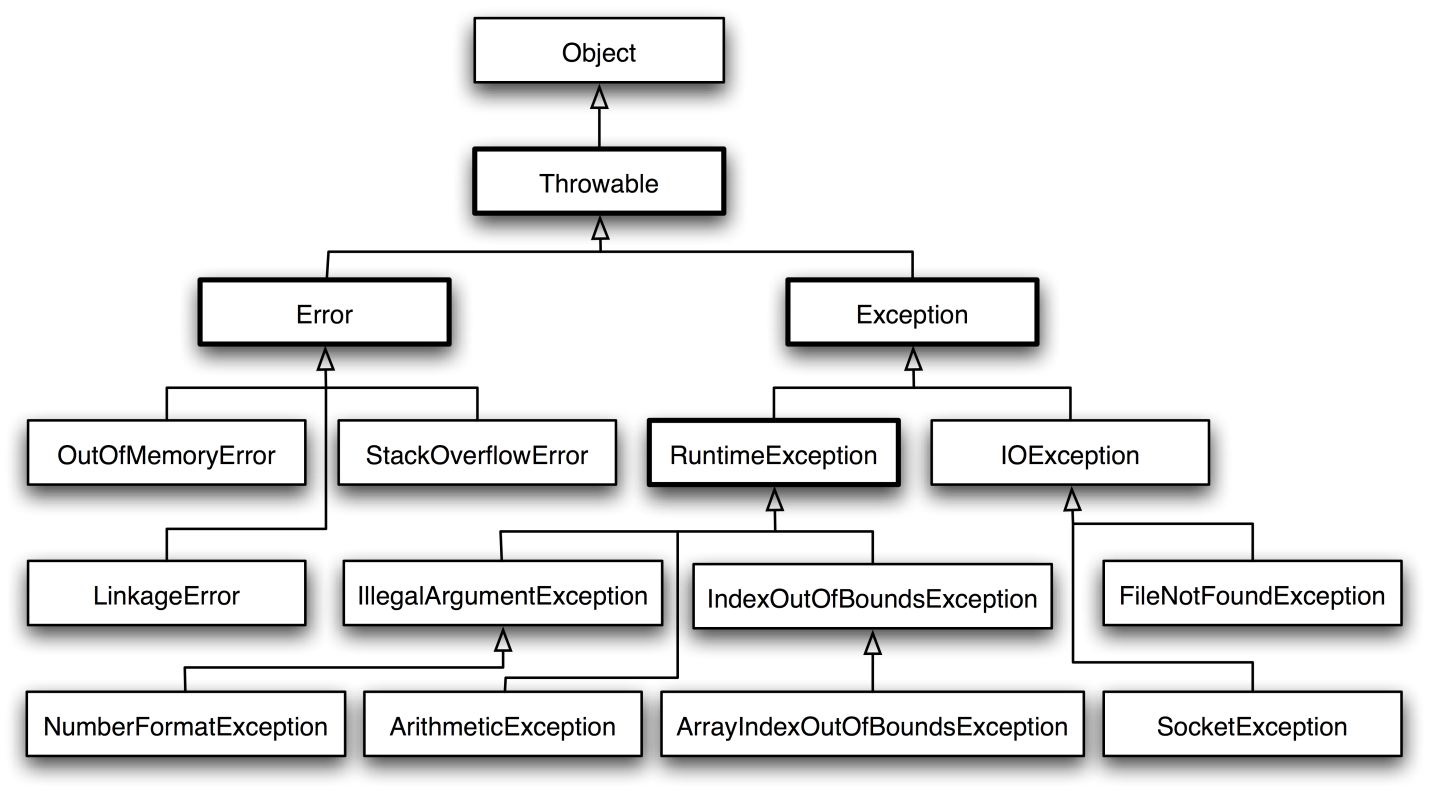
1. **What is an Exception?**
2. An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.

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

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

Exceptions come in two flavors: checked and unchecked.

* Checked exceptions include all subtypes of Exception, excluding classes that extend RuntimeException.
* Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using throws, or handle the exception with an appropriate try/catch.
* Subtypes of Error or RuntimeException are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them, or to declare them, but the compiler doesn't care one way or the other.

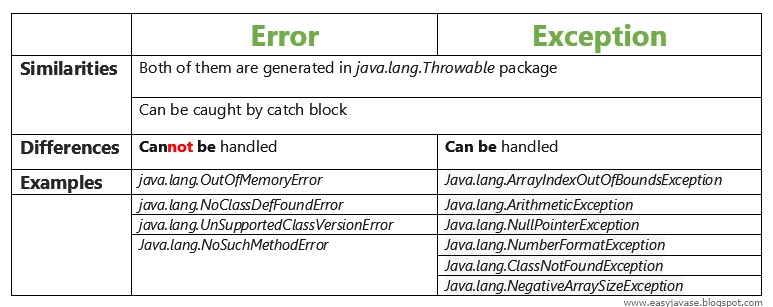
****

Important points for exceptions:

* If you use an optional finally block, it will always be invoked, regardless of whether an exception in the corresponding try is thrown or not, and regardless of whether a thrown exception is caught or not.
* The only exception to the finally-will-always-be-called rule is that a finally will not be invoked if the JVM shuts down. That could happen if code from the try or catch blocks calls **System.exit().**
* Just because finally is invoked does not mean it will complete. Code in the finally block could itself raise an exception or issue a System.exit().
* Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to main(), and main() is "ducking" the exception by declaring it).
* You can create your own exceptions, normally by extending Exception or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception.
* **All catch blocks must be ordered from most specific to most general**. If you have a catch clause for both IOException and Exception, you must put the catch for IOException first in your code. Otherwise, the IOException would be caught by catch(Exception e), because a catch argument can catch the specified exception or any of its subtypes! **The compiler will stop you from defining catch clauses that can never be reached.**
* Some exceptions are created by programmers, some by the JVM.

1. **What is the difference between error and an exception?**
2. An error is an irrecoverable condition occurring at runtime. Such as OutOfMemory error. Exceptions are conditions that occur because of bad input etc. e.g. FileNotFoundException will be thrown if the specified file does not exist.

Subtypes of Error or RuntimeException are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them, or to declare them, but the compiler doesn't care one way or the other.

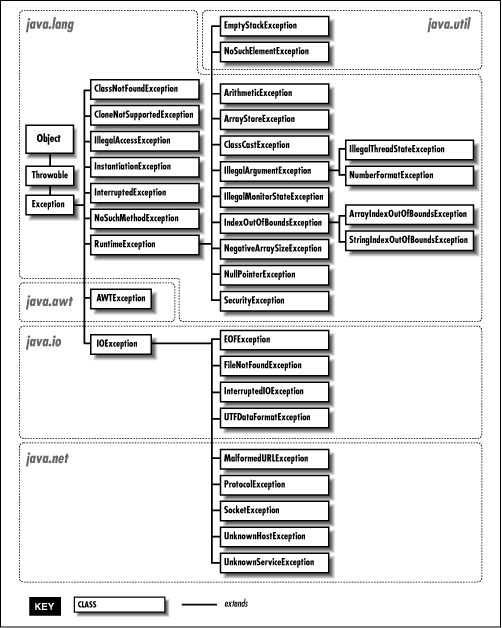


1. **Is it necessary that each try block must be followed by a catch block?**
2. It is not necessary that each try block must be followed by a catch block. It should be followed by either a catch block or a finally block.
3. **Explain Runtime Exceptions?**
4. It is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
5. **Which are the two subclasses under Exception class?**
6. The Exception class has two main subclasses : IOException class and RuntimeException Class.
7. **What is NullPointerException?**

A:  A NullPointerException is thrown when calling the instance method of a null object, accessing or modifying the field of a null object etc.

1. **Does it matter in what order catch statements for FileNotFoundException and IOException are written?**

**A:**  Yes, it does. The FileNotFoundException is inherited from the IOException. Exception's subclasses have to be caught first.



1. **When is the ArrayStoreException thrown?**
2. Thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects. For example, the following code generates an ArrayStoreException:

Object x[] = new String[3];

x[0] = new Integer(0);

1. **When ArithmeticException is thrown?**
2. The ArithmeticException is thrown when integer is divided by zero or taking the remainder of a number by zero. **It is never thrown in floating-point operations.**

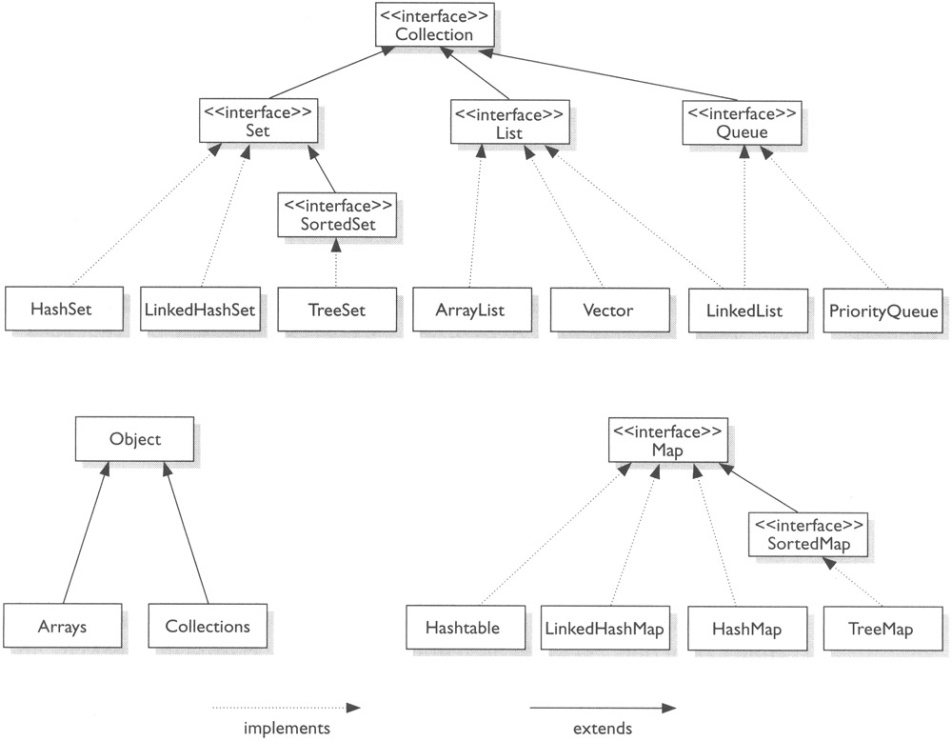
Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class. ArithmeticException objects may be constructed by the virtual machine as if suppression was disabled and/or the stack trace was not writable.

1. **What class of exceptions is generated by the Java run-time system?**
2. The Java runtime system generates RuntimeException and Error exceptions.
3. **What things should be kept in mind while creating your own exceptions in Java?**
4. While creating your own exception:

* All exceptions must be a child of Throwable.
* If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
* You want to write a runtime exception, you need to extend the RuntimeException class.

**XC:Collection**

1. **What is the Collections API?**
2. The Collections API is a set of classes and interfaces that support operations on collections of objects.

****



1. **What are the advantages of ArrayList over arrays?**
2. ArrayList can grow dynamically and provides more powerful insertion and search mechanisms than arrays.
3. **Why deletion in LinkedList is fast than ArrayList?**
4. The ArrayList class is a wrapper class for an array. It contains an inner array.

public ArrayList<T> {

private Object[] array;

private int size;

}

A LinkedList is a wrapper class for a linked list, with an inner node for managing the data.

public LinkedList<T> {

class Node<T> {

T data;

Node next;

Node prev;

}

private Node<T> first;

private Node<T> last;

private int size;

}

The ArrayList must move all the elements from array[index] to array[index-1] starting by the item to delete index. The LinkedList should navigate until that item and then erase that node by decoupling it from the list.

Because when you insert a new element and the array is full, you need to create a new array with more size (you can calculate the new size with a formula like 2 \* size or 3 \* size / 2). The LinkedList just add a new node next to the last.

1. **How do you decide when to use ArrayList and LinkedList?**
2. If you need to frequently add and remove elements from the middle of the list and only access the list elements sequentially, then LinkedList should be used. If you need to support random access, without inserting or removing elements from any place other than the end, then ArrayList should be used.
3. **What is a Values Collection View?**
4. It is a collection returned by the values() method of the Map Interface, It contains all the objects present as values in the map.

HashMap newmap = new HashMap();

// populate hash map

newmap.put(1, "tutorials");

newmap.put(2, "point");

newmap.put(3, "is best");

// checking collection view of the map

System.out.println("Collection view is: "+ newmap.values());

**Result:**

Collection view is: [tutorials, point, is best]

1. **What It Means If You Don't Override equals()**
   * 1. When you really need to know if two references are identical, use ==. But when you need to know if the objects themselves (not the references) are equal, use the equals() method.

The equals() method in class Object uses only the == operator for comparisons, so unless you override equals(), two objects are considered equal only if the two references refer to the same object.

Imagine you have a car, a very specific car (say, John's red Subaru Outback as opposed to Mary's purple Mini) that you want to put in a HashMap (a type of hashtable we'll look at later in this chapter), so that you can search on a particular car and retrieve the corresponding Person object that represents the owner. So you add the car instance as the key to the HashMap (along with a corresponding Person object as the value).

But now what happens when you want to do a search? You want to say to the HashMap collection, "Here's the car, now give me the Person object that goes with this car." But now you're in trouble unless you still have a reference to the exact object you used as the key when you added it to the Collection. *In other words, you can't make an identical Car object and use it for the search.*

The bottom line is this: if you want objects of your class to be used as keys for a hashtable (or as elements in any data structure that uses equivalency for searching for—and/or retrieving—an object), then you must override equals() so that two different instances can be considered the same. So how would we fix the car? You might override the equals() method so that it compares the unique VIN (Vehicle Identification Number) as the basis of comparison. That way, you can use one instance when you add it to a Collection, and essentially re-create an identical instance when you want to do a search based on that object as the key. Of course, overriding the equals() method for Car also allows the potential that more than one object representing a single unique car can exist, which might not be safe in your design. Fortunately, the String and wrapper classes work well as keys in hashtables—they override the equals() method. So rather than using the actual car instance as the key into the car/owner pair, you could simply use a String that represents the unique identifier for the car. That way, you'll never have more than

one instance representing a specific car, but you can still use the car—or rather, one of the car's attributes—as the search key.

1. public boolean equals(Object o) {

2. if ((o instanceof Moof) && (((Moof)o).getMoofValue()== this.moofValue)) {

3. return true;

4. } else {

5. return false;

6. }

7. }

Be sure that the object being tested is of the correct type! It comes in polymorphically as type Object, so you need to do an instanceof test on it. Having two objects of different class types be considered equal is usually not a good idea, but that's a design issue we won't go into here. Besides, you'd still have to do the instanceof test just to be sure that you could cast the object argument to the correct type so that you can access its methods or variables in order to actually do the comparison. Remember, if the object doesn't pass the instanceof test, then

you'll get a runtime ClassCastException.

***Remember that the*** equals()***,*** hashCode()***, and*** toString() ***methods are all*** public***. The following would not be a valid override of the*** equals() ***method, although it might appear to be if you don’t look closely enough during the exam:***

class Foo { boolean equals(Object o) { } } 🡪 Default equal method is not valid

***And watch out for the argument types as well. The following method is an overload, but not an override of the*** equals() ***method:***

class Boo { public boolean equals(Boo b) { } } 🡪 Object type Boo makes it overloaded

1. **What are equals() contract?**
2. Here are the contracts:
3. It is **reflexive**. For any reference value x, x.equals(x) should return true.
4. It is **symmetric**. For any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
5. It is **transitive**. For any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
6. It is **consistent**. For any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
7. For any non-null reference value x, x.equals(null) should return false.
8. **What It Means If You Don't Override Hashcode()**
9. Hashcodes are typically used to increase the performance of large collections of data. The hashcode value of an object is used by some collection classes. It is kind of an object ID number, it isn't necessarily unique. Collections such as HashMap and HashSet use the hashcode value of an object to determine how the object should be *stored* in the collection, and the hashcode is used again to help *locate* the object in the collection.

**Understanding Hashcodes**

In order to understand what's appropriate and correct, we have to look at how some of the collections use hashcodes.

Imagine a set of buckets lined up on the floor. Someone hands you a piece of paper with a name on it. You take the name and calculate an integer code from it by using A is 1, B is 2, and so on, and adding the numeric values of all the letters in the name together.

Key Hashcode Algorithm Hashcode

Alex A(1) + L(12) + E(5) + X(24) = 42

Bob B(2) + O(15) + B(2) = 19

Dirk D(4) + I(9) + R(18) + K(11) = 42

Fred F(6) + R(18) + E(5) + D(4) = 33

We don't introduce anything random, we simply have an algorithm that will always run the same way given a specific input, so the output will always be identical for any two identical inputs. So far so good? Now the way you use that code (and we'll call it a hashcode now) is to determine which bucket to place the piece of paper into (imagine that each bucket represents a different code number you might get). Now imagine that someone comes up and shows you a name and says, "Please retrieve the piece of paper that matches this name." So you look at the name they show you, and run the same hashcode-generating algorithm. The hashcode tells you in which bucket you should look to find the name. You might have noticed a little flaw in our system, though. Two different names might result in the same value. For example, the names Amy and May have the same letters, so the hashcode will be identical for both names. That's acceptable, but it does mean that when someone asks you (the bucket-clerk) for the Amy piece of paper, you'll still have to search through the target bucket reading each name until we find Amy rather than May. The hashcode tells you only which bucket to go into, but not how to locate the name once we're in that bucket.

***1. Find the right bucket (using*** hashCode()***)***

***2. Search the bucket for the right element (using*** equals() ***).***

So for efficiency, your goal is to have the papers distributed as evenly as possible across all buckets. Ideally, you might have just one name per bucket so that when someone asked for a paper you could simply calculate the hashcode and just grab the one paper. The least efficient (but still functional) hashcode generator would return the same hashcode (say, 42) regardless of the name, so that all the papers landed in the same bucket while the others stood empty.

When you put an object in a collection that uses hashcodes, the collection uses the hashcode of the object to decide in which bucket/slot the object should land. As long as the object (stored in the collection, like a paper in the bucket) you're trying to search for has the same hashcode as the object you're using for the search (the name you show to the person working the buckets), then the object will be found.

Now can you see why if two objects are considered equal, their hashcodes must also be equal? Otherwise, you'd never be able to find the object since the default hashcode method in class Object virtually always comes up with a unique number for each object, even if the equals() method is overridden in such a way that two or more objects are considered equal. It doesn't matter how equal the objects are if their hashcodes don't reflect that. So one more time: If two objects are equal, their hashcodes must be equal as well.

**Implementing hashCode()**

Your hashCode()implementation should use the same instance variables. Here's an example:

class HasHash {

public int x;

HasHash(int xVal) { x = xVal; }

public boolean equals(Object o) {

HasHash h = (HasHash) o;//Don't try at home without instanceof test

if (h.x == this.x) {

return true;

} else {

return false;

}

}

public int hashCode() { return (x \* 17); }

}

This equals() method says two objects are equal if they have the same x value, so objects with the same x value will have to return identical hashcodes.

1. **What are hashCode() contract?**
2. Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
3. If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result.
4. It is NOT required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.
5. **Explain TreeSet?**

**A:**  TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.

**import** java.util.Iterator;

**import** java.util.TreeSet;

**public** **class** TreeSetExample {

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

System.out.println("Tree Set Example!**\n**");

TreeSet<Integer> tree = **new** TreeSet<Integer>();

tree.add(12);

tree.add(63);

tree.add(34);

tree.add(45);

*// here it test it's sorted, 63 is the last element. see output below*

Iterator<Integer> iterator = tree.iterator();

System.out.print("Tree set data: ");

*// Displaying the Tree set data*

**while** (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

System.out.println();

*// Check empty or not*

**if** (tree.isEmpty()) {

System.out.print("Tree Set is empty.");

} **else** {

System.out.println("Tree Set size: " + tree.size());

}

*// Retrieve first data from tree set*

System.out.println("First data: " + tree.first());

*// Retrieve last data from tree set*

System.out.println("Last data: " + tree.last());

**if** (tree.remove(45)) { *// remove element by value*

System.out.println("Data is removed from tree set");

} **else** {

System.out.println("Data doesn't exist!");

}

System.out.print("Now the tree set contain: ");

iterator = tree.iterator();

*// Displaying the Tree set data*

**while** (iterator.hasNext()) {

System.out.print(iterator.next() + " ");

}

System.out.println();

System.out.println("Now the size of tree set: " + tree.size());

*// Remove all*

tree.clear();

**if** (tree.isEmpty()) {

System.out.print("Tree Set is empty.");

} **else** {

System.out.println("Tree Set size: " + tree.size());

}

}

}

Tree Set Example!

Tree set data: 12 34 45 63

Tree Set size: 4

First data: 12

Last data: 63

Data is removed from tree set

Now the tree set contain: 12 34 63

Now the size of tree set: 3

Tree Set is empty.

1. **What is Comparable Interface?**
2. When we say Comparable (please note the ‘able’ in the word), it actually means that the interface enables or gives the classes (which implements it) the ability to compare their instances with each other.

Also, the interface provides a method compareTo which takes a parameter T and returns an integer. This means that if a class implements this interface, it has to provide implementation for the compareTo method as well. Now the logic inside can be anything, anything at all. You are free to write your own comparison logic and say how the object makes a comparison with other objects of the same kind (I say same kind because, its pointless to compare two things of different kind, something like comparing a train and a dog might not make sense).

<http://techieme.in/comparable-and-comparato/>

1. **What are various methods in queue interface?**

|  |  |
| --- | --- |
| **Insert** | |
| **add** | **offer** |
| boolean add(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available. **Specified by:** add in interface Collection<E> Parameters: e - the element to add **Returns:** true (as specified by Collection.add(E)) **Throws:** IllegalStateException - if the element cannot be added at this time due to capacity restrictions ClassCastException - if the class of the specified element prevents it from being added to this queue NullPointerException - if the specified element is null and this queue does not permit null elements IllegalArgumentException - if some property of this element prevents it from being added to this queue | boolean offer(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to add(E), which can fail to insert an element only by throwing an exception. **Parameters:** e - the element to add **Returns:** true if the element was added to this queue, else false **Throws:** ClassCastException - if the class of the specified element prevents it from being added to this queue NullPointerException - if the specified element is null and this queue does not permit null elements IllegalArgumentException - if some property of this element prevents it from being added to this queue |
| **Delete** | |
| **remove** | **poll** |
| E remove() Retrieves and removes the head of this queue. This method differs from poll only in that it throws an exception (NoSuchElementException) if this queue is empty. | E poll() Retrieves and removes the head of this queue, or returns null if this queue is empty. |
| **Read** | |
| **element** | **peek** |
| E element() Retrieves, but does not remove, the head of this queue. This method differs from peek only in that it throws an exception(NoSuchElementException) if this queue is empty. | E peek() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

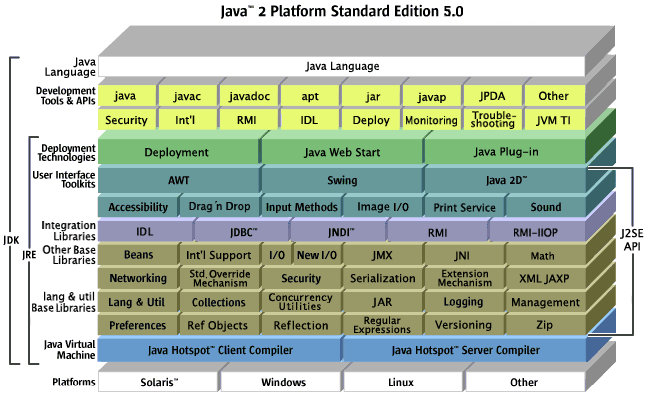
**XP:Packaging**

1. **Define JRE i.e. Java Runtime Environment?**
2. When asked typical Java Interview Questions most startup Java developers get confused with JDK and JRE. And eventually, they settle for ‘anything would do man, as long as my program runs!!’ Not quite right if you aspire to make a living and career out of Programming.

The “JDK” is the Java Development Kit. i.e., the JDK is bundle of software that you can use to develop Java based software.

The “JRE” is the Java Runtime Environment. I.e., the JRE is an implementation of the Java Virtual Machine which actually executes Java programs.

Typically, each JDK contains one (or more) JRE’s along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc.



1. **Define Packages in Java?**
2. A **package is a namespace that organizes a set of related classes and interfaces**. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.
3. **Why Packages are used?**
4. Packages are used in Java in-order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations, etc., easier.
5. **What is JAR file?**
6. The Java Archive (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications.

The JAR file format provides many benefits:

* ***Security***: You can digitally sign the contents of a JAR file. Users who recognize your signature can then optionally grant your software security privileges it wouldn't otherwise have.
* ***Decreased download time***: If your applet is bundled in a JAR file, the applet's class files and associated resources can be downloaded to a browser in a single HTTP transaction without the need for opening a new connection for each file.
* ***Compression***: The JAR format allows you to compress your files for efficient storage.
* ***Packaging for extensions***: The extensions framework provides a means by which you can add functionality to the Java core platform, and the JAR file format defines the packaging for extensions. By using the JAR file format, you can turn your software into extensions as well.
* ***Package Sealing*:** Packages stored in JAR files can be optionally sealed so that the package can enforce version consistency. Sealing a package within a JAR file means that all classes defined in that package must be found in the same JAR file.
* ***Package Versioning***: A JAR file can hold data about the files it contains, such as vendor and version information in (/META-INF/MANIFEST.MF).
* ***Portability***: The mechanism for handling JAR files is a standard part of the Java platform's core API.

**An entire directory tree structure can be archived in a single JAR file.**

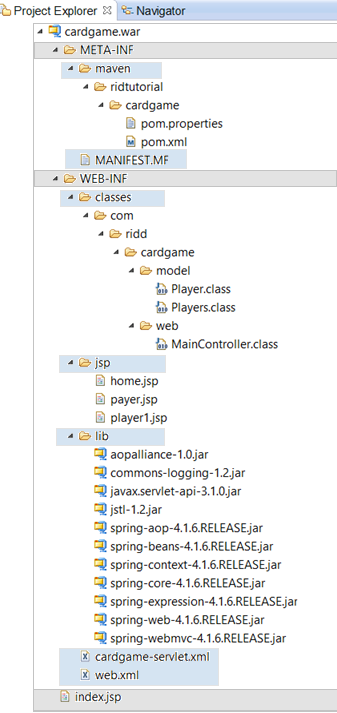
* JAR files can be searched by java and javac.
* When you include a JAR file in a classpath, you must include not only the directory in which the JAR file is located, but the name of the JAR file too.

1. **What is a WAR file?**
2. WAR is collection of JavaServer Pages, Java Servlets, Java classes, XML files, static Web pages etc.

WAR (Web Archive) is a module that goes into web container of Java EE application server. A JavaEE application server has two containers (runtime environments) - one is a web container and the other is a EJB container.

The Web container hosts web applications based on JSP/Servlets API - designed specifically for web request handling - more of request/response distributed computing. Web container requires the web module to be packaged in WAR file that is a special JAR file with a web.xml file in the WEB-INF folder.

EJB container hosts enterprise java beans based on EJB API designed to provide extended business functionality such as declarative transactions, declarative method level security and multiprotocol support - more of RPC style of distributed computing. EJB container required EJB module to be packaged in JAR file having ejb-jar.xml file in META-INF folder.



1. **What is an EAR file?**
2. Ear files (files with a .ear extension) is a standard JAR file (and therefore a Zip file) with a .ear extension.

All above files (.jar and .war) are packaged as JAR file with .ear (enterprise archive) extension and deployed into Application Server.

An EAR file contains all of the components that make up a particular J2EE application.

Enterprise application may consist of one or more modules that can either be Web modules (packaged in WAR file) or EJB modules (packaged in JAR file) or both of them. Enterprise applications are packaged in EAR file that is a special JAR file containing an application.xml file in the META-INF folder.

EAR may consists of

* Web module has a .war extension
* POJO Java classes may be deployed in .jar files.
* An Enterprise Java Bean module has a .jar extension.
* A Resource Adapter module has a .rar extension.

# XT:Thread

1. **What do you mean by Multithreaded program?**
2. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

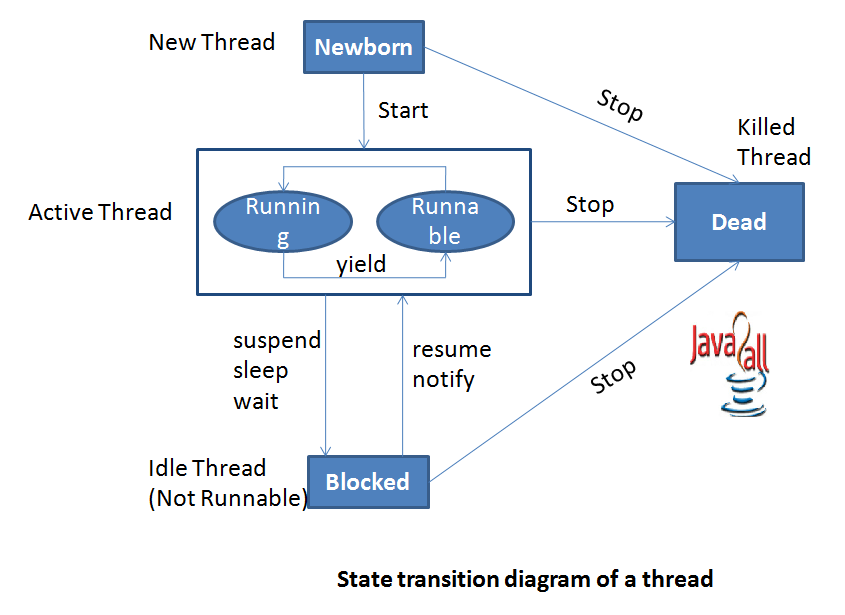
By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

1. **How can a dead thread be restarted?**
   1. A dead thread cannot be restarted.

When the execution of run() method is over, as the job it is meant is done, it is brought to dead state. It is done implicitly by JVM. In dead state, the thread object is garbage collected. It is the end of the life cycle of thread. Once a thread is removed, it cannot be restarted again (as the thread object does not exist). This state can be compared with destroy() method of applets.

A thread can be killed and brought to dead state, anytime from any state, by calling explicitly stop() method.



- See more at: <http://way2java.com/multithreading/life-cycle-of-thread/#sthash.RRP5ZbH8.dpuf>

1. **When a thread is created and started, what is its initial state?**
2. **Initial state**: A program has created a thread's thread object, but the thread does not yet exist because the thread object's start() method has not yet been called.

**NEW:** A thread that has not yet started is in this state.

**Runnable state**: This is a thread's default state. After the call to start() completes, a thread becomes runnable whether or not that thread is running, that is, using the processor. Although many threads might be runnable, only one currently runs. Thread schedulers determine which runnable thread to assign to the processor.

**Blocked state**: When a thread executes the sleep(), wait(), or join() methods, when a thread attempts to read data not yet available from a network, and when a thread waits to acquire a lock, that thread is in the blocked state: it is neither running nor in a position to run. (You can probably think of other times when a thread would wait for something to happen.) When a blocked thread unblocks, that thread moves to the runnable state.

* [WAITING](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#WAITING)  
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
* [TIMED\_WAITING](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#TIMED_WAITING)  
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.

**Terminating state**: Once execution **leaves a thread's run() method**, that thread is in the terminating state. In other words, the thread ceases to exist.

1. **What are the two ways in which Thread can be created?**
2. Thread can be created by: implementing Runnable interface, extending the Thread class.

* ***Provide a Runnable object.*** The [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) interface defines a single method run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the [HelloRunnable](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloRunnable.java) example:

public class HelloRunnable implements Runnable {

public void run() {

System.out.println("Hello from a thread!");

}

public static void main(String args[]) {

(new Thread(new HelloRunnable())).start();

}

}

* ***Subclass Thread.***The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the [HelloThread](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloThread.java" \t "_blank)example:

public class HelloThread extends Thread {

public void run() {

System.out.println("Hello from a thread!");

}

public static void main(String args[]) {

(new HelloThread()).start();

}

}

The **Java Virtual Machine calls the run method** of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

1. **What's the difference between the methods sleep() and wait()?**

|  |  |  |
| --- | --- | --- |
|  | Wait | Sleep |
| Definition | It is a method **in Object class**. It makes the current thread into the "Not Runnable" state. Wait is called on an object, not a thread. Before calling wait() method, the object should be synchronized, means the object should be inside synchronized block. The **call to wait() releases the acquired lock.** | It is a static method **in Thread class**. It makes the current thread into the "Not Runnable" state for specified amount of time. During this time, the **thread keeps the lock (monitors) it has acquired.** |
| Synchonized | wait **should be called from synchronized context** i.e. from block or method, If you do not call it using synchronized context, it will throw IllegalMonitorStateException | It **need not be called from synchronized block or methods** |
| Calls on | wait method **operates on Object** and defined in Object class | Sleep method **operates on current thread** and is in java.lang.Thread |
| Release of lock | wait release lock of object on which it is called and also other locks if it holds any | Sleep method does not release lock at all |
| Wake up condition | until call notify() or notifyAll() from Object class | Until time expires or calls interrupt() |
| static | wait is non static method | sleep is static method |

<http://www.logicaltrinkets.com/wordpress/?p=153>

<http://howtodoinjava.com/2013/03/08/difference-between-sleep-and-wait/>

<http://www.programcreek.com/2009/02/notify-and-wait-example/>

<http://www.tutorialspoint.com/java/lang/thread_sleep_millis.htm>

**public** **class** ThreadA {

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

ThreadB b = **new** ThreadB();

b.start();

**synchronized**(b){

**try**{

System.out.println("Waiting for b to complete...");

b.wait();

}**catch**(InterruptedException e){

e.printStackTrace();

}

System.out.println("Total is: " + b.total);

}

}

}

**class** ThreadB **extends** Thread{

**int** total;

@Override

**public** **void** run(){

**synchronized**(**this**){

**for**(**int** i=0; i<100 ; i++){

total += i;

}

notify();

}

}

}

1. **What are the ways in which a thread can enter the waiting state?**
   1. A thread can enter the waiting state by :

* A running thread may enter a blocked/waiting state by a wait(), sleep(), or join() call.
* A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
* It can also enter the waiting state by invoking its (deprecated) suspend() method.

When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will go directly from waiting to running (well, for all practical purposes anyway).

1. **How does multi-threading take place on a computer with a single CPU?**
   1. The operating system's task scheduler allocates execution time to multiple tasks. By quickly switching between executing tasks, it creates the impression that tasks execute sequentially.
2. **What invokes a thread's run() method?**
3. After a thread is started, via its start() method of the Thread class, the JVM invokes the thread's run() method when the thread is initially executed.

So what happens after you call start().The good stuff:

1. A new thread of execution starts (with a new call stack).
2. The thread moves from the new state to the runnable state.
3. When the thread gets a chance to execute, its target run() method will run.

Be sure you remember the following: You start a Thread, not a Runnable. You call start() on a Thread instance, not on a Runnable instance. The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread:

class FooRunnable implements Runnable {

public void run() {

for(int x = 1; x < 6; x++) {

System.out.println("Runnable running");

}

}

}

public class TestThreads {

public static void main (String [] args) {

FooRunnable r = new FooRunnable();

Thread t = new Thread(r);

t.start();

}

}

So if you see code that calls the run() method on a Runnable (or even on a Thread instance), that’s perfectly legal. But it doesn’t mean the run() method will run in a separate thread! Calling a run() method directly just means you’re invoking a method from whatever thread is currently executing, and the run() method goes onto the current call stack rather than at the beginning of a new call stack. The following code does not start a new thread of execution:

Thread t = new Thread();

t.run(); // Legal, but does not start a new thread

1. **What is the difference between yielding and sleeping?**
2. When a task invokes its yield() method, it returns to the **ready** state. When a task invokes its sleep() method, it returns to the **waiting** state.

* Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
* A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep() method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
* The sleep() method is a static method that sleeps the currently executing thread's state. One thread cannot tell another thread to sleep.
* The yield() method may cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a different thread selected to run. A thread might yield and then immediately reenter the running state.

1. **What is synchronization?**
2. Synchronization is the capability to control the access of multiple threads to shared resources. Synchronized keyword in java provides locking which ensures mutual exclusive access of shared resource and prevent data race.
3. **What are synchronized methods and synchronized statements?**

**A:** Synchronized methods are methods that are used to control access to an object. A synchronized statement can only be executed after a thread has acquired the lock for the object or class referenced in the synchronized statement.

To make a method synchronized, simply add the synchronized keyword to its declaration:

public class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

public void addName(String name) {

synchronized(this) {

lastName = name;

nameCount++;

}

nameList.add(name);

}

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

1. **What do you mean by synchronized Non Access Modifier?**
2. Java provides these modifiers for providing functionalities other than Access Modifiers, synchronized used to indicate that a method can be accessed by only one thread at a time.

* **The synchronized modifier applies only to methods and code blocks.**
* synchronized methods can have any access control and can also be marked final.
* Instance variables can't be abstract, synchronized, native, or strictfp.
* **~~The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.~~**

# ~~Vector: It's like a slower ArrayList, but it has synchronized methods.~~

# ~~Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.~~

* A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.

# Synchronized methods prevent more than one thread from accessing an object's critical method code simultaneously.

# You can use the synchronized keyword as a method modifier, or to start a synchronized block of code.

# While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.

# Static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.

<http://www.javatpoint.com/static-synchronization-example>

**XIO: IO FILE handling**

1. **What is the purpose of File class?**
2. Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion etc.

File file = new File("fileWrite2.txt"); // create a File object AND

// open "fileWrite2.txt"

FileReader fr = new FileReader(file); // create a FileReader to get

// data from 'file'

BufferedReader br = new BufferedReader(fr); // create a BufferReader to

// get its data from a Reader

String data = br.readLine(); // read some data

BufferedReader in = new BufferedReader(new FileReader("foo.in"));

File file = new File("foo");

Always creates a File object, and then does one of two things:

1. If "foo" does NOT exist, no actual file is created.
2. If "foo" does exist, the new File object refers to the existing file.

Notice that File file = new File("foo"); NEVER creates an actual file.

There are two ways to create a file:

1. Invoke the createNewFile() method on a File object. For example:

File file = new File("foo"); // no file yet

file.createNewFile(); // make a file, "foo" which

// is assigned to 'file'

1. Create a Writer or a Stream. Specifically, create a FileWriter, a PrintWriter, or a FileOutputStream. Whenever you create an instance of one of these classes, you automatically create a file, unless one already exists, for instance

File file = new File("foo"); // no file yet

FileWriter fw = new FileWriter(file); // make a FileWriter object AND

// make a file, "foo" to which

// 'file' is assigned, AND assign

// 'fw' to the FileWriter

**File I/O**

* The classes you need to understand in java.io are File, FileReader, BufferedReader, FileWriter, BufferedWriter, PrintWriter, and Console.
* A new File object doesn't mean there's a new file on your hard drive.
* File objects can represent either a file or a directory.
* The File class lets you manage (add, rename, and delete) files and directories.
* The methods createNewFile() and mkdir() add entries to your file system.
* FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
* Classes in java.io are designed to be "chained" or "wrapped." (This is a common use of the decorator design pattern.)
* It's very common to "wrap" a BufferedReader around a FileReader or a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
* PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.

**File Streaming** File streaming is carried out by FileInputStream object instance in Java.

// read byte by byte and store into this variable

int byt;

while((byt = fileInputStream.read()) != -1) {

fileOutputStream.write(byt);

}

This objects actually reads a byte(8-bit) at a time and writes it to the given file. It's good when you are working with raw data file, such as images or audio (same but uses AudioInputStream object), etc. but for the text files, it is inconvenient or slower, because it waste the time looping though. you also need to provide a text file character set otherwise whatever you write you won't see in the file, you would just see some random text and signs.

**File Reading (basically streaming with character)** FileReader is used for file reading, so the operation is as,

int c;

while ( (c = fileReader.read()) != -1) { // some logic }

instead of fetching a byte each time, it fetches 2 bytes a time. But still runs the same number of times as the characters in your file. But it is better than byte steaming such as 'FileInputStream' when you're dealing with files. But the both instances use an integer variable to store the value coming from an input file. Byte stream stores 8 bits into a variable and FileReader stores 16 bits, but here, you don't have to specify file's character set and some few more properties, it automatically does it for you, and also supports internationalization. but it's still slower.

**Buffering streams** Now file buffering is faster than file stream reading. It's the same concept as you see on youtube. youtube buffers some amount of video before it actually starts playing it, so you can have flawless video watching experience.

So the file buffering uses BufferedReader object instance, and you need to specify which stream you want to buffer, in this case it's a file so you need to pass a FileReader object.

BufferedReader br = new BufferedReader( new FileReader("example.txt") );

now here, bufferedReader buffers the file means it keeps reading a file one by one character, until it hits '\n', '\r\n' (new line) character and stores fetched characters into a string (in short fetches the lines and stores it into buffer).

String line;

while ((line = br.readLine()) != null) {

printWriter.println(line);

}

now, you can see instead of storing each bytes or 2bytes value in a int variable, it stores the whole line into a memory and works on it. But buffer reader cannot work alone, because it just buffers so you need to specify which stream you want to buffer? It could be a file stream as you saw earlier or it could be a console, as

BufferedReader br = new BufferedReader(System.in);

then it will scan a whole line from the console. that's why you use bufferedReader.readLine() in java to collect the input from a console.

1. **What is the difference between the Reader/Writer class hierarchy and the InputStream/ OutputStream class hierarchy?**
2. First thing you need to understand is

* What is **STREAM?**

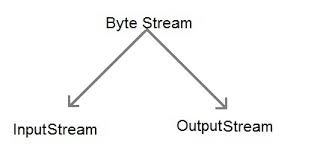
A stream can be defined as a sequence of data. The Input Stream is used to read data from a source and the Output Stream is used for writing data to a destination.

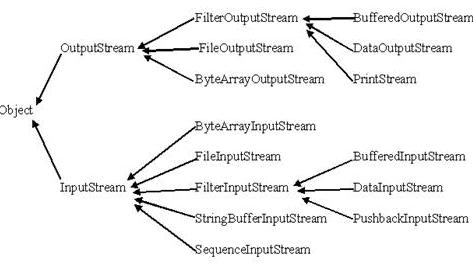
* *Next is type of streams*

We have byte stream and character stream.

**Byte Stream**

Classes we have in Input Stream and output stream



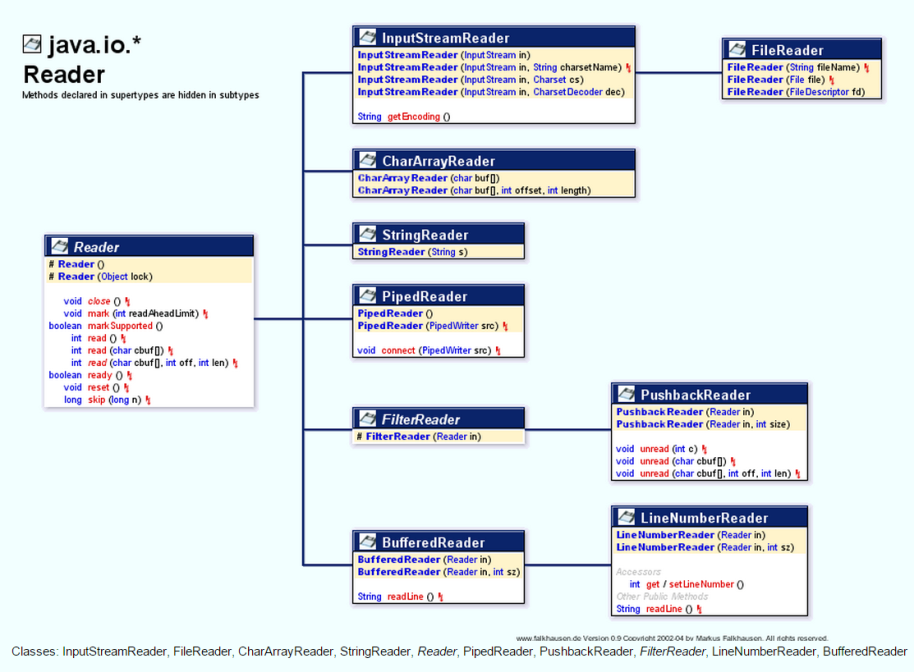


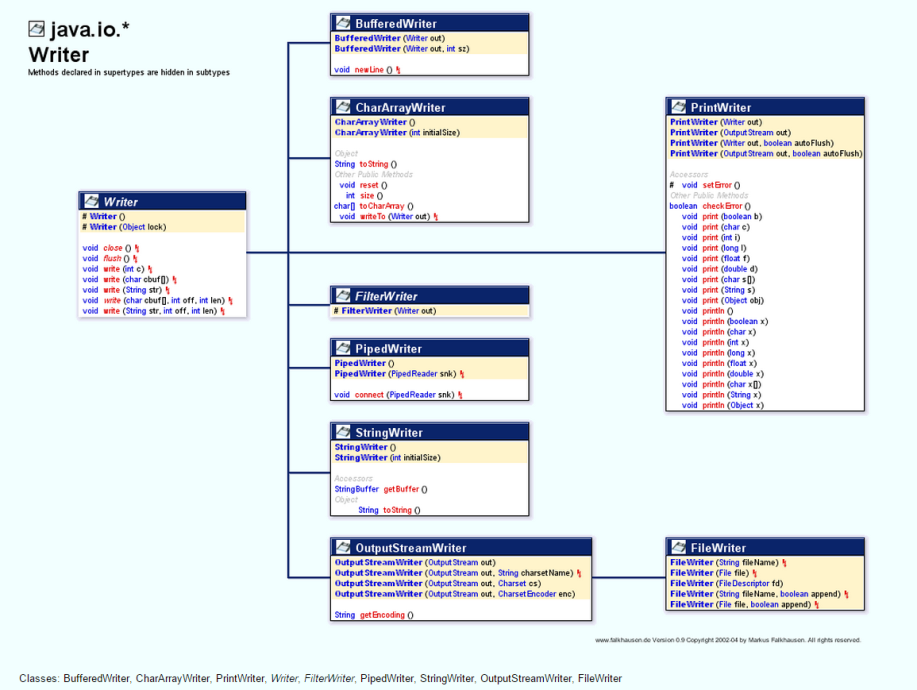
As the name suggests in **simple terms** input stream is used to input the data and output stream is used to output the data

Java **byte streams** are used to perform input and output of **8-bit bytes.** Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Also Java **Byte streams** are used to perform input and output of 8-bit bytes.

**Character Stream**

Java **Character streams** are used to perform input and output for **16-bit unicode**. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and fileWriter writes two bytes at a time.





**For reference**

1. [What is InputStream & Output Stream? Why do we use them and when do we use each of them?](http://stackoverflow.com/questions/1830698/what-is-inputstream-output-stream-why-do-we-use-them-and-when-do-we-use-each)
2. [java DataOutputStream getOutputStream() getInputStream()](http://stackoverflow.com/questions/4913559/java-dataoutputstream-getoutputstream-getinputstream)

**Example for getInputStream and getOutputStream**

1. <http://zerioh.tripod.com/ressources/sockets.html>

**New Link** <http://docs.oracle.com/javase/tutorial/essential/io/buffers.html>

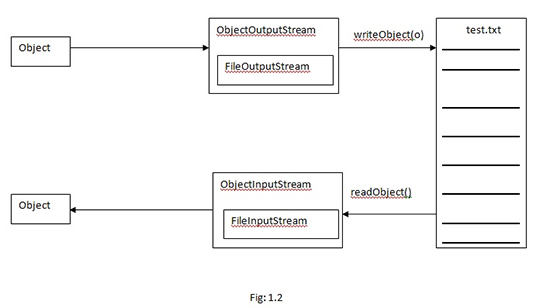
1. **What is the use of Serialization**
2. As written above serialization will translate the Object state to Byte Streams. This Byte stream can be used for different purpose.

* Write to Disk
* Store in Memory
* Sent byte stream to other platform over network
* Save byte stream in DB(As BLOB)

1. **What is Serialization and deserialization?**
2. Serialization is the process of writing the state of an object to a byte stream. Deserialization is the process of restoring these objects.

As byte stream create is platform neutral hence once objects created in one system can be deserialized in other platform.

* The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
* A class must implement Serializable before its objects can be serialized.
* The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method deserializes objects.
* If you mark an instance variable transient, it will not be serialized even thought the rest of the object's state will be.
* You can supplement a class's automatic serialization process by implementing the writeObject() and readObject() methods. If you do this, embedding calls to defaultWriteObject() and defaultReadObject(), respectively, will handle the part of serialization that happens normally.
* If a superclass implements Serializable, then its subclasses do automatically.
* If Super class is not serializable then when sub class is de serialized then super class’s default constructor will be invoked. Hence all variable will get default value and reference will be null.



1. **What is a transient variable?**
2. A transient variable is a variable that may not be serialized during Serialization and which is initialized by its default value during de-serialization.

# XG: Garbage Collection

1. **Explain garbage collection in Java?**

**A:**  Java objects reside in an area called the heap. The heap is created when the JVM starts up and may increase or decrease in size while the application runs. When the heap becomes full, garbage is collected. During the garbage collection objects that are no longer used are cleared, thus making space for new objects. Any object created in the heap space has global access and can be referenced from anywhere of the application.

<https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html>

<https://docs.oracle.com/javase/specs/jvms/se5.0/html/Overview.doc.html#1732>

<http://www.journaldev.com/4098/java-heap-memory-vs-stack-memory-difference>

1. **Does garbage collection guarantee that a program will not run out of memory?**

**A:** Garbage collection does not guarantee that a program will not run out of memory. It is possible for programs to use up memory resources faster than they are garbage collected. It is also possible for programs to create objects that are not subject to garbage collection.

<https://weblogs.java.net/blog/2006/05/04/understanding-weak-references>

1. **What is finalize() method?**
2. It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called finalize( ), and it can be used to ensure that an object terminates cleanly.

* Class Object has a finalize() method.
* Remember that any code that you can put into a normal method you can put into finalize().
* The finalize() method is guaranteed to run once and only once before the garbage collector deletes an object.
* The garbage collector makes no guarantees, finalize() may never run. So, any code that you put into your class's overridden finalize() method is not guaranteed to run.
* You can uneligibilize an object for GC from within finalize(). You override the finalized method and pass the object that's eligible for gc to another method, object is saved from deletion.

1. **Q: Which number is denoted by leading 0x or 0X in java?**

**A:** Hexadecimal Numbers are denoted by leading 0x or 0X in java, example: 0XF

1. **Q: Where import statement is used in a Java program?**

**A:** Import statement is allowed at the beginning of the program file after package statement.

1. **Q: Explain suspend() method under Thread class>**

**A:** It is used to pause or temporarily stop the execution of the thread.

1. **Q: Explain isAlive() method under Thread class?**

**A:** It is used to find out whether a thread is still running or not.

1. **Q: What is currentThread()?**

**A:** It is a public static method used to obtain a reference to the current thread.

1. **Q: Explain main thread under Thread class execution?**

**A:** The main thread is created automatically and it begins to execute immediately when a program starts. It is a thread from which all other child threads originate.

1. **Q: Why Generics are used in Java?**

**A:** Generics provide compile-time type safety that allows programmers to catch invalid types at compile time. Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types.

1. **Q: What environment variables do I need to set on my machine in order to be able to run Java programs?**

**A:** CLASSPATH and PATH are the two variables.

1. **Q: Is there any need to import java.lang package?**

**A:** No, there is no need to import this package. It is by default loaded internally by the JVM.

1. **Q: What is Nested top-level class?**

**A:** If a class is declared within a class and specify the static modifier, the compiler treats the class just like any other top-level class. Nested top-level class is an Inner class.

1. **Q: What is Externalizable interface?**

**A:** Externalizable is an interface which contains two methods readExternal and writeExternal. These methods give you a control over the serialization mechanism.

1. **Q: If System.exit (0); is written at the end of the try block, will the finally block still execute?**

**A:** No in this case the finally block will not execute because when you say System.exit (0); the control immediately goes out of the program, and thus finally never executes.

1. **Q: What is daemon thread?**

**A:** Daemon thread is a low priority thread, which runs intermittently in the back ground doing the garbage collection operation for the java runtime system.

1. **Q: Which method is used to create the daemon thread?**

**A:** setDaemon method is used to create a daemon thread.

1. **Q: Which method must be implemented by all threads?**

**A:** All tasks must implement the run() method

1. **Q: What is the GregorianCalendar class?**

**A:** The GregorianCalendar provides support for traditional Western calendars

1. **Q: What is the SimpleTimeZone class?**

**A:** The SimpleTimeZone class provides support for a Gregorian calendar .

1. **Q: What is the difference between the size and capacity of a Vector?**

**A:** The size is the number of elements actually stored in the vector, while capacity is the maximum number of elements it can store at a given instance of time.

1. **Q: Can a vector contain heterogenous objects?**

**A:** Yes a Vector can contain heterogenous objects. Because a Vector stores everything in terms of Object.

1. **Q: What is an enumeration?**

**A:** An enumeration is an interface containing methods for accessing the underlying data structure from which the enumeration is obtained. It allows sequential access to all the elements stored in the collection.

1. **Q: What is difference between Path and Classpath?**

**A:** Path and Classpath are operating system level environment variales. Path is defines where the system can find the executables(.exe) files and classpath is used to specify the location of .class files.

1. **Q: Can a class declared as private be accessed outside it's package?**

**A:** No, it's not possible to accessed outside it's package.

1. **Q: What are the restriction imposed on a static method or a static block of code?**

**A:** A static method should not refer to instance variables without creating an instance and cannot use "this" operator to refer the instance.

1. **Q: Can an Interface extend another Interface?**

**A:** Yes an Interface can inherit another Interface, for that matter an Interface can extend more than one Interface.

1. **Q: What is an object's lock and which object's have locks?**

**A:** An object's lock is a mechanism that is used by multiple threads to obtain synchronized access to the object. A thread may execute a synchronized method of an object only after it has acquired the object's lock.

1. **Q: What is Downcasting?**

**A:** It is the casting from a general to a more specific type, i.e. casting down the hierarchy.

1. **Q: What are order of precedence and associativity and how are they used?**

**A:** Order of precedence determines the order in which operators are evaluated in expressions. Associatity determines whether an expression is evaluated left-to-right or right-to-left.

1. **Q: If a method is declared as protected, where may the method be accessed?**

**A:** A protected method may only be accessed by classes or interfaces of the same package or by subclasses of the class in which it is declared.

1. **Q: What is the difference between inner class and nested class?**

**A:** When a class is defined within a scope of another class, then it becomes inner class. If the access modifier of the inner class is static, then it becomes nested class.

1. **Q: What is constructor chaining and how is it achieved in Java?**

**A:** A child object constructor always first needs to construct its parent. In Java it is done via an implicit call to the no-args constructor as the first statement.

1. **Q: Can a double value be cast to a byte?**

**A:** Yes, a double value can be cast to a byte.

1. **Q: How does a try statement determine which catch clause should be used to handle an exception?**

**A:** When an exception is thrown within the body of a try statement, the catch clauses of the try statement are examined in the order in which they appear. The first catch clause that is capable of handling the exception is executed. The remaining catch clauses are ignored.

1. **Q: What will be the default values of all the elements of an array defined as an instance variable?**

**A:** If the array is an array of primitive types, then all the elements of the array will be initialized to the default value corresponding to that primitive type.

1. **How many bits are used to represent Unicode, ASCII, UTF-16, and UTF-8 characters?**

**A:** Unicode requires 16 bits

ASCII requires 7 bits.

ASCII character set uses only 7 bits, It is usually represented as 8 bits.

UTF-8 represents characters using 8, 16, and 18 bit patterns.

UTF-16 uses 16-bit and larger bit patterns.

1. **What are use cases?**

**A:**  A use case is a methodology used in system analysis to identify, clarify, and organize system requirements. The use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal. It consists of a group of elements (for example, classes and interfaces) that can be used together in a way that will have an effect larger than the sum of the separate elements combined. The use case should contain all system activities that have significance to the users. A use case can be thought of as a collection of possible scenarios related to a particular goal, indeed, the use case and goal are sometimes considered to be synonymous.

1. **What is Dynamic Binding(late binding)?**

**A:**  Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.

<http://javarevisited.blogspot.com.au/2012/03/what-is-static-and-dynamic-binding-in.html>

1. **Q: Define Network Programming?**

**A:** It refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

1. **Q: What is a Socket?**

**A:** Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

1. **Q: Advantages of Java Sockets?**

**A:** Sockets are flexible and sufficient. Efficient socket based programming can be easily implemented for general communications. It cause low network traffic.

1. **Q: Disadvantages of Java Sockets?**

**A:** Socket based communications allows only to send packets of raw data between applications. Both the client-side and server-side have to provide mechanisms to make the data useful in any way.

1. **Q: Which class is used by server applications to obtain a port and listen for client requests?**

**A:** java.net.ServerSocket class is used by server applications to obtain a port and listen for client requests

1. **Q: Which class represents the socket that both the client and server use to communicate with each other?**

**A:** java.net.Socket class represents the socket that both the client and server use to communicate with each other.