In **Java 1.2** Swing and Collection framework was added and suspend (), resume() and stop()methods were deprecated from **Thread** class.

**Java 1.4** contained several important changes. Keyword assert, chained exceptions and channel based I/O System was introduced.

**Java 1**.**5** was called **J2SE** **5**, it added following major new features:

1. Generics
2. Annotations
3. Autoboxing and auto-unboxing
4. Enumerations
5. For-each Loop
6. Varargs
7. Static Import
8. Formatted I/O
9. Concurrency utilities

**Java SE 7** which included many new changes, like:

1. Now String can be used to control Switch statement.
2. Multi Catch Exception
3. try-with-resource statement
4. Binary Integer Literals
5. Underscore in numeric literals, etc.

**Java SE 8,** it was released on March 18, 2014. Some of the major new features introduced in JAVA 8 are,

1. Lambda Expressions
2. New Collection Package java.util.stream to provide Stream API.
3. Enhanced Security
4. Nashorn Javascript Engine included
5. Parallel Array Sorting
6. The JDBC-ODBC Bridge has been removed etc.

Key Features of Java:

1. **Object Oriented**

In java everything is Object which has some data and behavior. Java can be easily extended as it is based on Object Model.

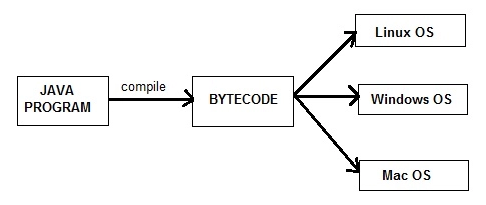
1. **Robust**

Java tries to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking. Main areas which Java improved were Memory Management and mishandled Exceptions by introducing automatic Garbage Collector and Exception Handling

1. **Platform Independent**

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.



#### Secure

Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

#### Multi-Threading

Java multithreading feature makes it possible to write program that can do many tasks simultaneously. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

#### Portable

Java Byte code can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

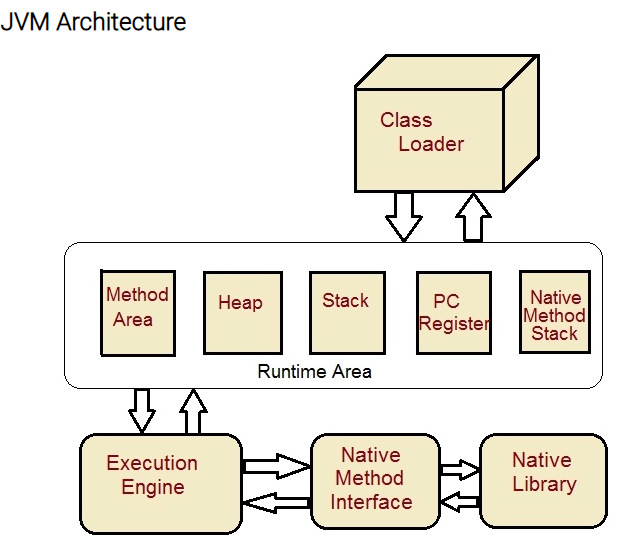
#### High Performance

Java is an interpreted language, so it will never be as fast as a compiled language like C or C++. But, Java enables high performance with the use of just-in-time compiler.

### **What is JVM?**

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your \*.java files to obtain \*.class files that contain the bytecodes understandable by the JVM

JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap.



**Class Loader:** Class loader loads the Class for execution.

**Method area:** Stores pre-class structure as constant pool.

**Heap:** Heap is in which objects are allocated.

**Stack:** Local variables and partial results are store here. Each thread has a private JVM stack created when the thread is created.

**Program register:** Program register holds the address of JVM instruction currently being executed.

**Native method stack:** It contains all native used in application.

**Executive Engine:** Execution engine controls the execute of instructions contained in the methods of the classes.

**Native Method Interface:** Native method interface gives an interface between java code and native code during execution.

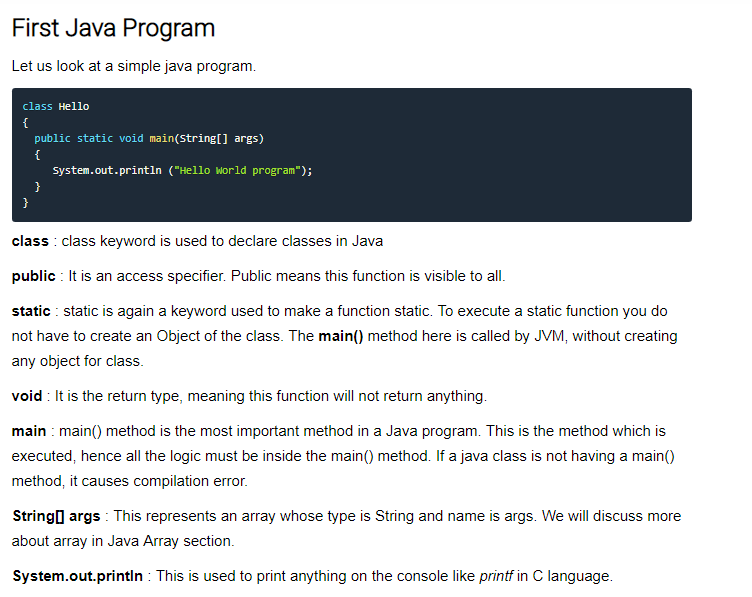
**Native Method Libraries:** Native Libraries consist of files required for the execution of native code.

**Difference between JDK and JRE:**

**JRE**: The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.

**JDK**: The JDK also called Java Development Kit is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.



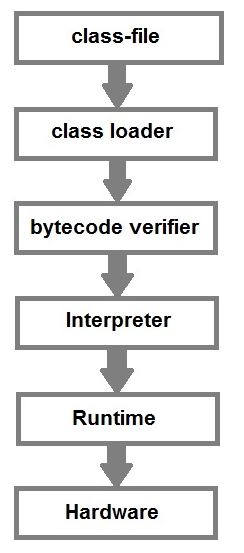


#### What happens at Runtime

1. Class loader loads the java class

2. Byte Code verifier checks the code fragments for illegal codes that can violate access right to the object.

3. Interpreter reads the byte code stream and then executes the instructions, step by step.



### **Data Types in Java**

1. **Primitive Data type**

**Integer**

1. Byte: 1-byte Value range from -128 to 127
2. Short: 2-byte Value range from -32768 to 32767
3. Int: 4-bytes Value range from -2147483648 to 2147483647
4. Long:8-byte

**Floating-Point Number**

1. **float:** It is 4 bytes(32-bits) float data type. Default value 0.0f. example: float ff=10.3f;
2. **double:** It is 8 bytes(64-bits) float data type. Default value 0.0d. example: double db=11.123;

**Characters**

1. **char**: It is 2 bytes(16-bits) unsigned unicode character. Range 0 to 65,535. example: char c='a';

**Boolean**

boolean

**Type Casting:**

**Widening:**

int i = 100;

long l = i; //no explicit type casting required

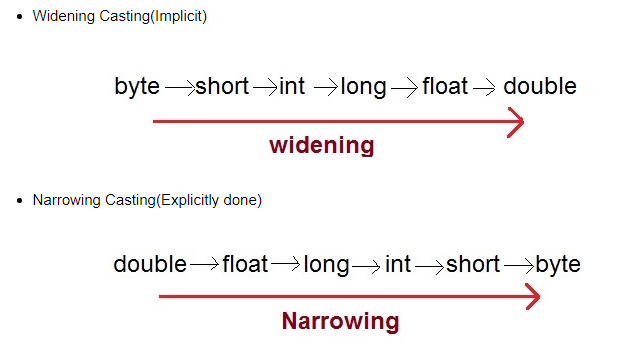
float f = l;

**Narrowing:**

double d = 100.04;

long l = (long)d; //explicit type casting required

int i = (int)l; //explicit type casting required



#### Autoboxing and Unboxing

* **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
* **Auto-Unboxing** is a process by which the value of an object is automatically extracted from a type Wrapper class.

#### Example of Autoboxing and Unboxing

class Test

{

public static void main(String[] args)

{

Integer iob = 100; //Auto-boxing of int i.e converting primitive data type int to a Wrapper class Integer

int i = iob; //Auto-unboxing of Integer i.e converting Wrapper class Integer to a primitve type int

System.out.println(i+" "+iob);

Character cob = 'a'; //Auto-boxing of char i.e converting primitive data type char to a Wrapper class Character

char ch = cob; //Auto-unboxing of Character i.e converting Wrapper class Character to a primitive type char

System.out.println(cob+" "+ch);

}

}

100 100

a a

**What is a variable?**

The naming of an address is known as variable. Variable is the name of memory location.

Java Programming language defines mainly three kinds of variables.

1. **Instance variables**

It is declared inside a class but outside any method, constructor or block. They are referred as object variable. Each object has its own copy of each variable.

1. **Static Variables**

Static are class variables declared with static keyword. Static variables are initialized only once. Static variables are also used in declaring constant along with final keyword.

Additional points on static variable:

* static variable is also known as class variable.
* static means to remain constant.
* In Java, it means that it will be constant for all the instances created for that class.
* static variable need not be called from object.
* It is called by classname.static variable name.

1. **Local Variables**

Local variables are initialized when method, constructor or block start and will be **destroyed** once its end. Local variable resides in **stack**.

**Array:**

An array is a collection of similar data types.  It gets memory in heap area.

Bitwise Operator:

&, |, ^, >> (left shift), << (Right Shift)

#### instanceOf operator

This operator is used for object reference variables. The operator checks whether the object is of type (class type or interface type).

#### Main Features of OOPS

1. Inheritence
2. Polymorphism
3. Encapsulation
4. Abstraction

#### Rules for Java Class

1. A class can have **only public or default** (no modifier) access specifier.
2. It can be either **abstract**, **final** or **concrete** (normal class).
3. It must have the class keyword, and class must be followed by a legal identifier.
4. It may optionally extend one parent class. By default, it will extend java.lang.Object.
5. It may optionally implement any number of comma-separated interfaces.
6. Each **.java** source file may contain only one public class. A source file may contain any number of default visible classes.
7. Finally, the source file name must match the public class name and it must have a .java suffix.

class Student.

{

String name;

int rollno;

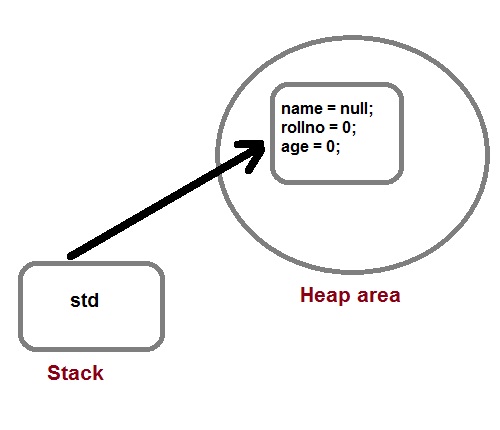
int age;

}

When a reference is made to a student with its property then it becomes an object, physical existence of Student class.

Student std=new Student ();

After the above statement **std** is instance/object of Student class. Here the **new** keyword creates an actual physical copy of the object and assign it to the **std** variable. It will have physical existence and get memory in heap area. **The new operator dynamically allocates memory for an object**



**Q How a class is initialized in java?**

A Class is initialized in Java when an instance of class is created using either **new** operator or using reflection using class.forName(). A class is also said to be initialized when a static method of **Class** is invoked, or a static field of **Class** is assigned.

**Q. How would you make a copy of an entire Java object with its state?**

Make that class implement **Cloneable** interface and call **clone ()** method on its object. **Clone ()** method is defined in **Object** class which is parent of all java class by default.

#### Q. Does constructors return any value?

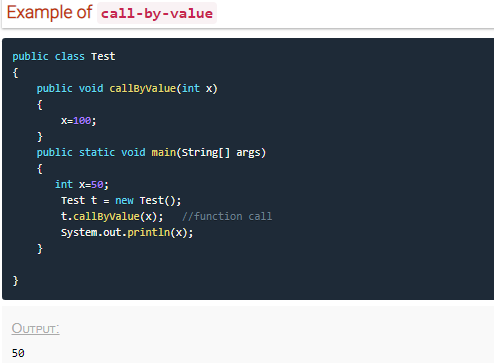
Yes, constructors return current instant of a class. But constructor signature cannot have any return type.

#### call-by-value and call-by-reference

There are two ways to pass an argument to a method

1. **call-by-value:** In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.
2. **call-by-reference:** In this reference of an argument is pass to a method. Any changes made inside the method will affect the argument value.

**NOTE:** There is only call by value in java, not call by reference.



**If two or more method have same name and same parameter list but differs in return type are not said to be overloaded method.**

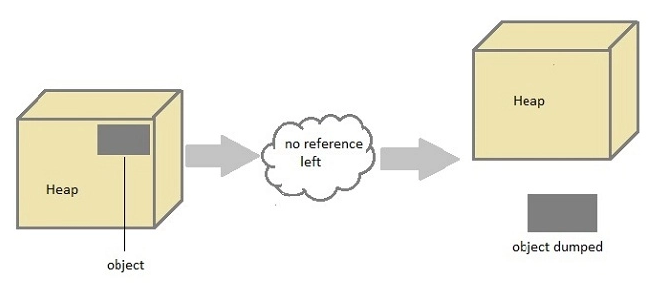
### **this keyword**

* **this** keyword is used to refer to current object.
* **this** is always a reference to the object on which method was invoked.
* **this** can be used to invoke current class constructor.
* **this** can be passed as an argument to another method.

### **Garbage Collection**

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called Garbage Collection. This is accomplished by the JVM.

Unlike C++ there is no explicit need to destroy object.



#### Can the Garbage Collection be forced explicitly?

No, the Garbage Collection cannot be forced explicitly. We may request JVM for **garbage collection** by calling **System.gc()** method. But This does not guarantee that JVM will perform the garbage collection.

#### Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

#### finalize () method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation, finalize () method is used. Finalize () method is called by garbage collection thread before collecting object. It’s the last chance for any object to perform cleanup utility.

#### Some Important Points to Remember

1. finalize() method is defined in **java.lang.Object** class, therefore it is available to all the classes.
2. finalize() method is declare as **proctected** inside Object class.
3. finalize() method gets called only once by a Daemon thread named GC (Garbage Collector)thread.

#### gc() Method

**gc()** method is used to call garbage collector explicitly. However **gc ()** method does not guarantee that JVM will perform the garbage collection. It only requests the JVM for garbage collection. This method is present in **System** and **Runtime** class.

#### Example for gc() method

public class Test

{

public static void main(String[] args)

{

Test t = new Test ();

t=null;

System.gc();

}

public void finalize ()

{

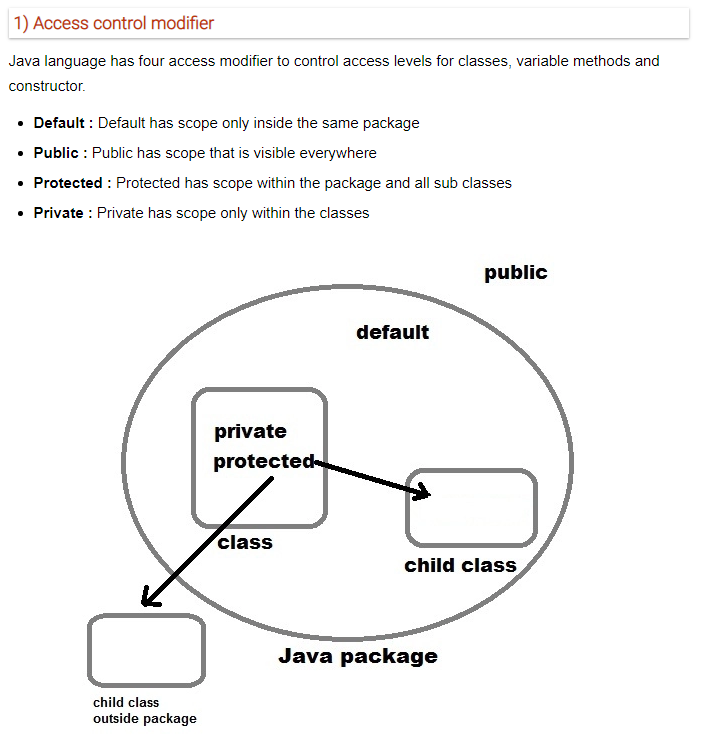
System.out.println("Garbage Collected");

}

}

Output: Garbage Collected

### **Modifiers in Java**



**Non-access Modifier**

1. **Final**

Final modifier is used to declare a field as final i.e. it prevents its content from being modified.

Final field **must be initialized** when it is declared.

Final keyword can be used with a variable, a method or a class.

1. **Final Variable:** When a variable is declared as final, then its value cannot be changed. The variable acts like a constant
2. **Final Method:** When a method is declared as final, it can be inherited/used in the subclass, but it cannot be overridden.
3. **Final Class:** A class can also be declared as final. A class declared as final cannot be inherited. **String** class in java.lang package is an example of final class. Method declared as final can be inherited but you cannot override(redefine) it
4. **Static:**

#### Static Modifier

Static Modifiers are used to create class variable and class methods which can be accessed without instance of a class.

#### Static with Variables

#### Static variable has only one single storage.

#### All the object of the class having static variable will have the same instance of static variable.

#### Static variables are initialized only once.

#### Saves memory.

class ST\_Employee

{

int eid;

String name;

static String company\_name ="StudyTonight";

public void show()

{

System.out.println(eid+" "+name+" "+company\_name);

}

public static void main( String[] args )

{

ST\_Employee se1 = new ST\_Employee();

se1.eid = 104;

se1.name = "Abhijit";

se1.show();

ST\_Employee se2 = new ST\_Employee();

se2.eid = 108;

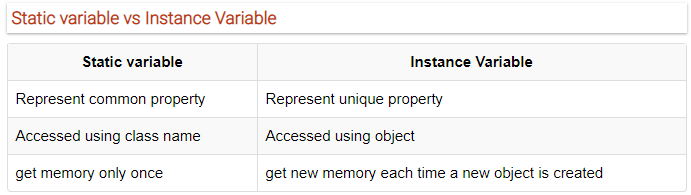
se2.name = "ankit";

se2.show();

}

}

#### 



#### Static Method:

#### A method can also be declared as static. Static methods do not need instance of its class for being accessed.

#### Static block

Static block is used to initialize static data member. Static block executes before main () method

#### Q. Why a non-static variable cannot be referenced from a static context?

When you try to access a non-static variable from a static context like main method, java compiler throws a message like "a non-static variable cannot be referenced from a static context".

This is because non-static variables are related with instance of class(object) and they **get created** when instance of a class is created by **using new operator**. So, if you try to access a non-static variable without any instance compiler will complain because those variables are not yet created, and they don't have any existence until an instance is created and associated with it.

#### Q. Why main () method is static in java?

Because static methods can be called without any instance of a class and **main ()** is called before any instance of a class is created.

#### Transient modifier

When an instance variable is declared as transient, then its value doesn't persist when an object is serialized

#### Synchronized modifier

When a method is synchronized it can be accessed by only one thread at a time.

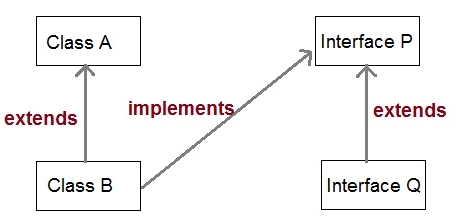
#### Volatile modifier

Volatile modifier tells the compiler that the volatile variable can be changed unexpectedly by other parts of your program. Volatile variables are used in case of multithreading program

### **Inheritance (IS-A)**

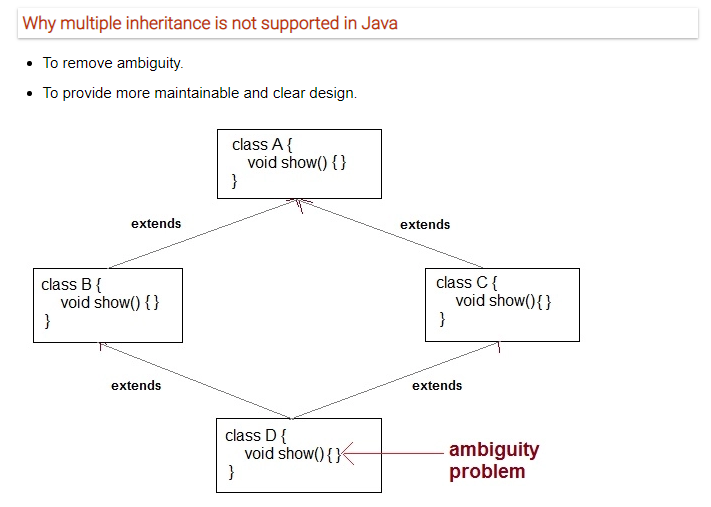
Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child) in Java language.

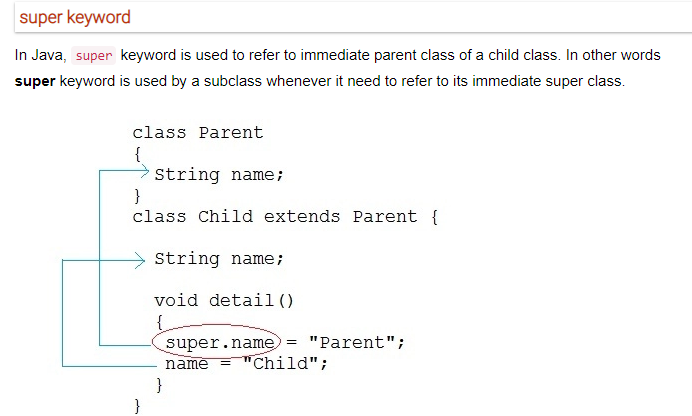
Inheritance defines **is-a** relationship between a Super class and its Sub class.  extends and implements keywords are used to describe inheritance in Java.



1. It promotes polymorphism by allowing method overriding.

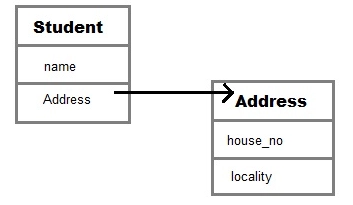
2. Main disadvantage of using inheritance is that the two classes (parent and child class) gets **tightly coupled**.





### **Aggregation (HAS-A)**

HAS-A relationship is based on usage, rather than inheritance. In other words, class A has-arelationship with class B, if code in class A has a reference to an instance of class B.



#### Q. What is Composition in java?

Composition is restricted form of Aggregation. Composition can be described as when one class which owns the other class, is destroyed then the other class cannot exist meaningfully without it. For example, a class **Car** cannot exist without **Engine**.

class Car

{

private Engine engine;

Car (Engine en)

{

engine = en;

}

}

#### Q. When to use Inheritance and Aggregation?

When you need to use property and behavior of a class without modifying it inside your class. In such case **Aggregation** is a better option. Whereas when you need to use and modify property and behavior of a class inside your class, its best to use **Inheritance**.

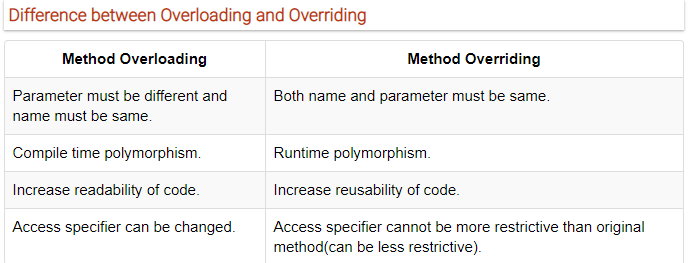
### **Method Overriding**

Method overriding is also referred to as runtime polymorphism. The key benefit of overriding is the ability to **define method that's specific to a subclass type**.

**Note:** Static methods cannot be overridden because, a static method is bounded with class whereas instance method is bounded with object.

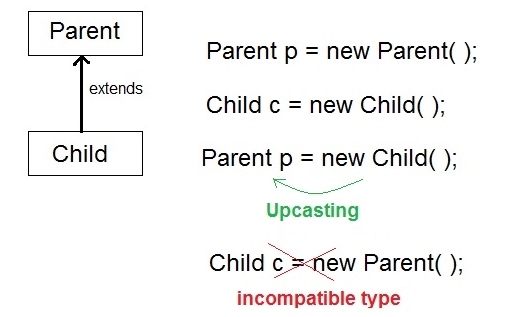
#### Covariant return type

Since Java 5, it is possible to override a method by changing its return type. If subclass override any method by changing the return type of super class method, then the return type of overridden method must be **subtype of return type** declared in original method inside the super class. This is the only way by which method can be overridden by changing its return type.



#### Q. Can we Override static method? Explain with reasons?

No, we cannot override static method. Because static method is bound to class whereas method overriding is associated with object i.e. at runtime.



When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**

|  |  |
| --- | --- |
| Static binding | Dynamic binding |
| Static binding in Java occurs during compile time | Dynamic binding occurs during runtime |
| Static binding uses type(Class) information for binding | Dynamic binding uses instance of class(Object) to resolve calling of method at run-time |
| Overloaded methods are bonded using static binding | Overridden methods are bonded using dynamic binding at runtime. |
| Static binding means when the type of object which is invoking the method is determined at compile time by the compiler | Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler |

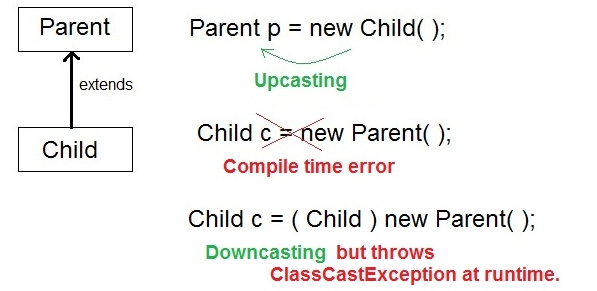
### **instance of operator**

In Java, instanceof operator is used to check the type of an object at runtime.

It is how our program can obtain run-time type information about an object.

instanceof operator is also important in case of casting object at runtime.

instanceof operator return Boolean value, if an object reference is of specified type then it returns **true** otherwise **false**.



class Parent{}

class Child1 extends Parent{}

class Child2 extends Parent{}

class Test

{

public static void main(String[] args)

{

Parent p = new Parent();

Child1 c1 = new Child1();

Child2 c2 = new Child2();

System.out.println(c1 instanceof Parent); //true

System.out.println(c2 instanceof Parent); //true

System.out.println(p instanceof Child1); //false

System.out.println(p instanceof Child2); //false

p = c1;

System.out.println(p instanceof Child1); //true

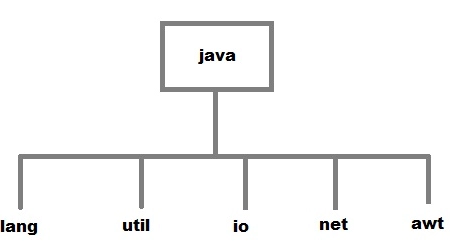
System.out.println(p instanceof Child2); //false

p = c2;

System.out.println(p instanceof Child1); //false

System.out.println(p instanceof Child2); //true }}

### **Java Package**



#### How to compile Java programs inside packages?

javac -d directory javafilename

**Example:**

javac -d . FirstProgram.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like **d:/abc** (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

### **Abstract class**

If a class contain any abstract method, then the class is declared as abstract class. An abstract class is never instantiated.

**Syntax:**

abstract class class\_name { }

#### Abstract method

Method that are declared without any body within an abstract class are called abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

**Syntax:**

abstract return\_type function\_name (); //No definition

1. Abstract classes can have Constructors, Member variables and Normal methods.
2. Abstract classes are never instantiated.
3. When you extend Abstract class with abstract method, you must define the abstract method in the child class or make the child class abstract.

#### When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy and use its subclasses to provide implementation details of the abstract class.

Ex:

abstract class Vehicle

{

public abstract void engine();

}

public class Car extends Vehicle {

public void engine()

{

System.out.println("Car engine");

//car engine implementation

}

public static void main(String[] args)

{

Vehicle v = new Car ();

v.engine();

}

}

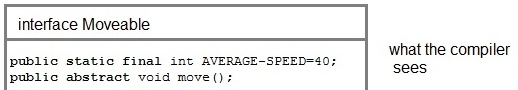
Here by casting instance of **Car** type to **Vehicle** reference, we are hiding the complexity of **Car** type under **Vechicle**. Now the **Vehicle** reference can be used to provide the implementation, but it will hide the actual implementation process.

### **Interface**

Interface is a pure abstract class.

Cannot create instance of an Interface and their methods are declared without any body. Interface is used to achieve complete abstraction in Java.





**Note:**Compiler automatically converts methods of Interface as public and abstract, and the data members as public, static and final by default.

#### Rules for using Interface

* Methods inside Interface must not be static, final, native or strictfp.
* All variables declared inside interface are implicitly public static final variables(constants).
* All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.

**Difference between an interface and an abstract class?**

| **Abstract class** | **Interface** |
| --- | --- |
| Abstract class is a class which contain one or more abstract methods, which must be implemented by its sub classes. | Interface is a Java Object containing method declaration but no implementation. The classes which implement the Interfaces must provide the method definition for all the methods. |
| Abstract class is a Class prefix with an abstract keyword followed by Class definition. | Interface is a pure abstract class which starts with interface keyword. |
| Abstract class can also contain concrete methods. | Whereas, Interface contains all abstract methods and final variable declarations. |
| Abstract classes are useful in a situation that Some general methods should be implemented, and specialization behavior should be implemented by child classes. | Interfaces are useful in a situation that all properties should be implemented. |

### **Nested Class**

A class defined within another class is known as Nested class. The scope of the nested class is bounded by the scope of its enclosing class.

#### Static Nested Class

If the nested class i.e. the class defined within another class, has static modifier applied in it, then it is called as static nested class. Since it is, static nested classes can access only static members of its outer class i.e. it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested class is rarely used.

#### Non-static Nested class

Non-static Nested class is the most important type of nested class. It is also known as **Inner class**. It has access to all variables and methods of Outer class including its private data members and methods and may refer to them directly.

#### Example of Inner class instantiated outside Outer class

class Outer

{

int count;

public void display ()

{

Inner in=new Inner ();

in.show();

}

class Inner {

public void show () {

System.out.println ("Inside inner "+(++count));

}

}

}

class Test

{

public static void main (String [] args)

{

Outer ot = new Outer ();

Outer.Inner in= ot.new Inner ();

in.show();

}

}

O/p: Inside inner 1

#### Anonymous class

A class without any name is called **Anonymous class**.

interface Animal

{

void type();

}

public class ATest {

public static void main (String args[])

{

Animal an = new Animal () { //Annonymous class created

public void type ()

{

System.out.println("Anonymous animal");

}

};

an.type();

}

}

O/p: Anonymous animal

Here a class is created which implements **Animal** interface and its name will be decided by the compiler. This anonymous class will provide implementation of **type ()** method.

**String**

String objects are immutable that means once a string object is created it cannot be altered.

#### What is an Immutable object?

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

#### Creating an Immutable class

public final class MyString

{

final String str;

MyString(String s)

{

this.str = s;

}

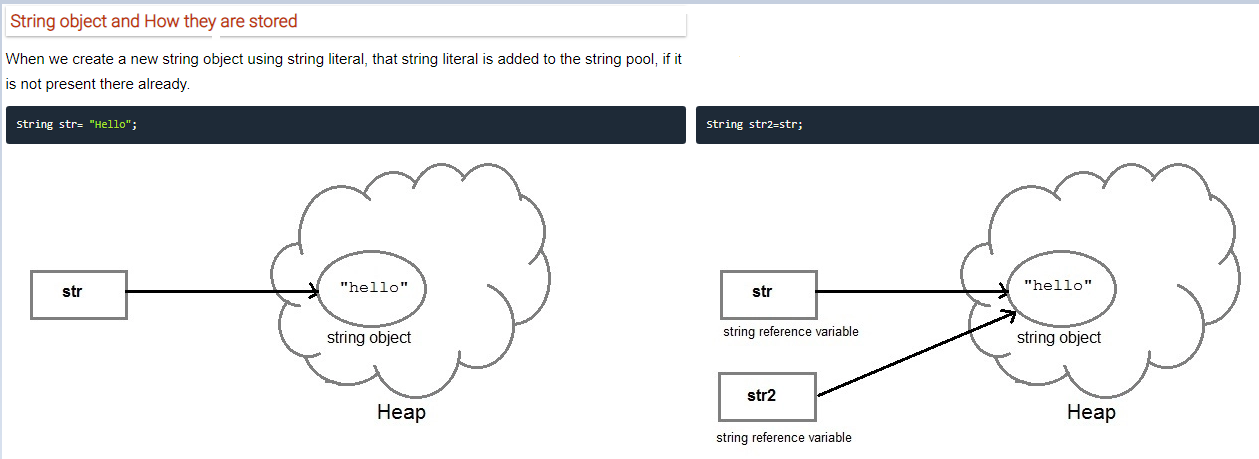
public String get()

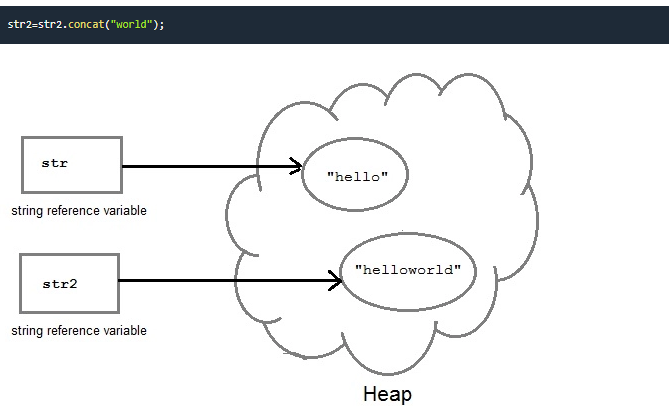
{

return str;

}

}





#### Using equals () method

equals () method compares two strings for equality

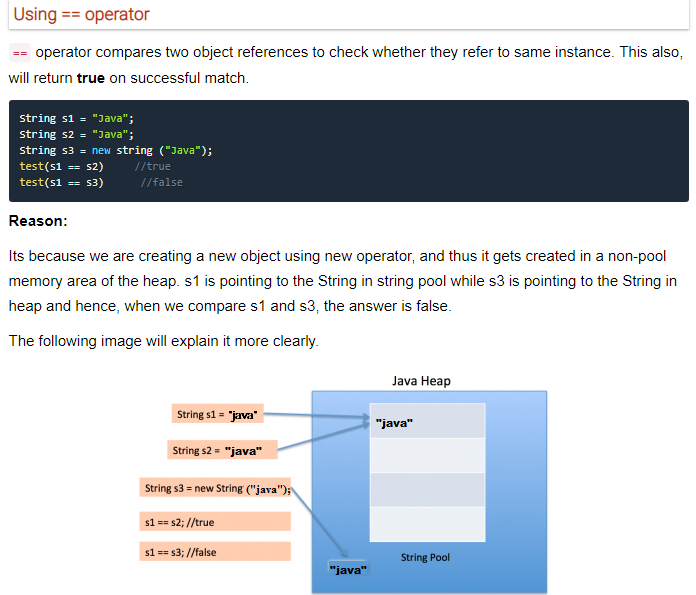
String s = "Hell";

String s1 = "Hello";

String s2 = "Hello";

s1.equals (s2); //true

s.equals(s1) ; //false



#### toString()

toString() method returns the string representation of the object used to invoke this method. toString() is used to represent any Java Object into a meaningful string representation. It is declared in the **Object class**, hence can be overridden by any java class. (Object class is super class of all java classes.)

public class Car {

public static void main(String args[])

{

Car c = new Car();

System.out.println(c);

}

public String toString()

{

return "This is my car object";

}

}

O/p: This is my car object

**If we don't override the toString() method and directly print the object, then it would print the object id.**

**StringBuffer class**

StringBuffer class is used to create a **mutable** string object i.e. its state can be changed after it is created.

So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also thread safe i.e. multiple threads cannot access it simultaneously.

#### Example showing difference between String and StringBuffer

**class Test {**

public static void main(String args[])

{

String str = "study";

str.concat("tonight");

System.out.println(str); // Output: study

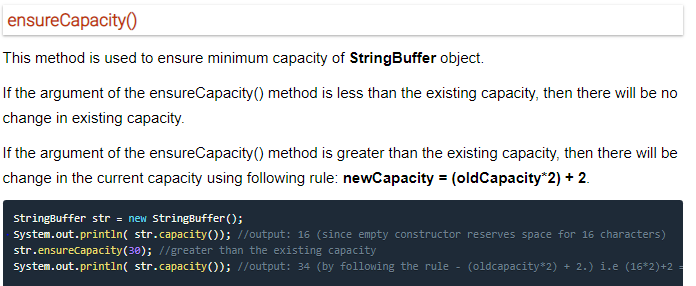
StringBuffer strB = new StringBuffer("study");

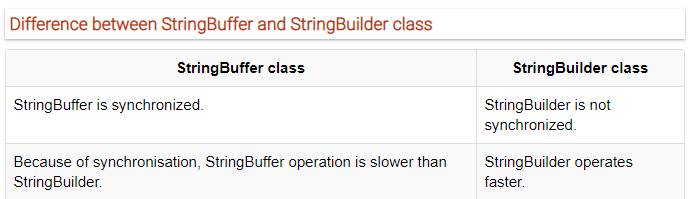
strB.append("tonight");

System.out.println(strB); // Output: studytonight

}

}





**Exception Handling**

#### Exception are categorized into 3 category.

* **Checked Exception**

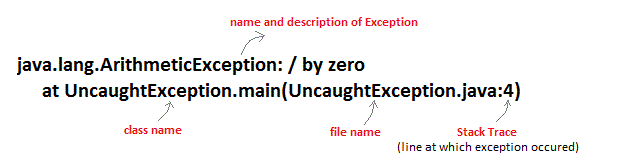
The exception that can be predicted by the programmer at the compile time. Example: File that need to be opened is not found. These types of exceptions must be checked at compile time.

* **Unchecked Exception**

Unchecked exceptions are the class that extends RuntimeException. Unchecked exception is ignored at compile time. Example: **ArithmeticException**, **NullPointerException**, **ArrayIndexOutOfBound** exception. Unchecked exceptions are checked at runtime.

* **Error**

Errors are typically ignored in code because you can rarely do anything about an error. Example:if stack overflow occurs, an error will arise. This type of error cannot be handled in the code.

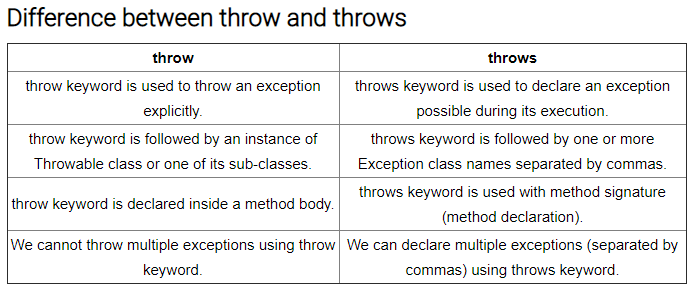


### **Try with Resource Statement**

This **try statement** contains a parenthesis in which one or more resources is declared. Any object that implements java.lang.AutoCloseable or java.io.Closeable, can be passed as a parameter to **try statement**. A resource is an object that is used in program and must be closed after the program is finished. The **try-with-resources statement** ensures that each resource is closed at the end of the statement of the try block. You do not have to explicitly close the resources.

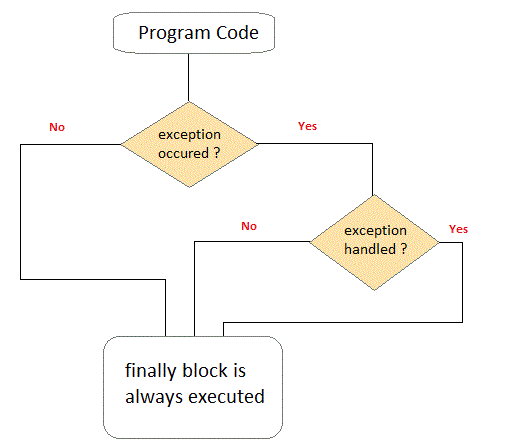
****

1. Any object that implements java.lang.AutoCloseable or java.io.Closeable can be passed as a parameter to try statement.
2. All the resources declared in the try-with-resources statement will be closed automatically when the try block exits. There is no need to close it explicitly.
3. We can write more than one resources in the try statement.



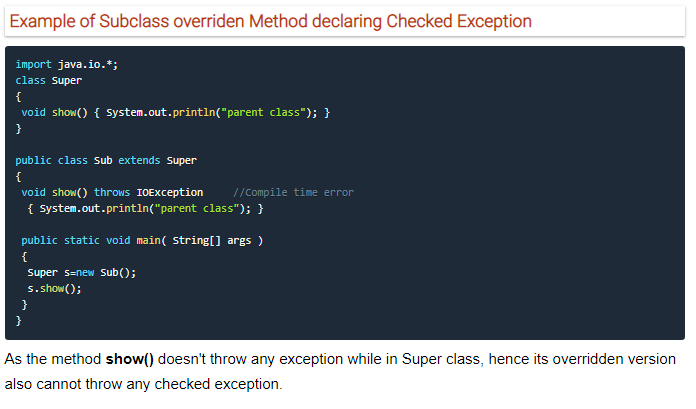
### finally clause

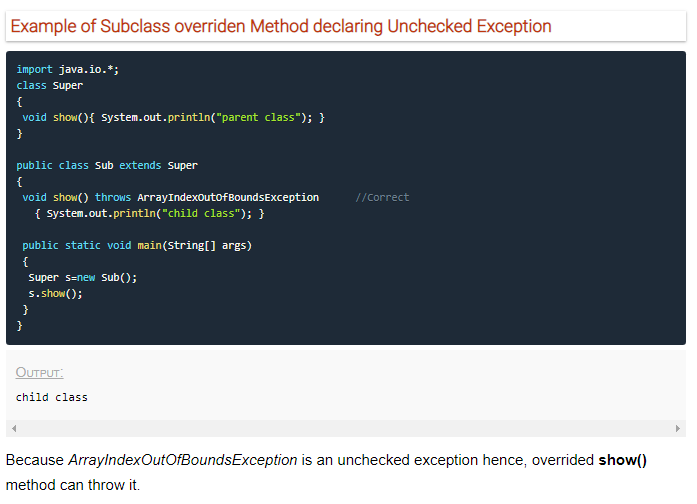
A finally keyword is used to create a block of code that follows a try block. A finally block of code is always executed whether an exception has occurred or not. Using a finally block, it lets you run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.



### **Method Overriding with Exception Handling**

If super class method does not declare any exception, then sub class overridden method cannot declare checked exception but it can declare unchecked exceptions.





#### More about Overridden Methods and Exceptions

If Super class method throws an exception, then Subclass overridden method can throw the same exception or no exception but must not throw parent exception of the exception thrown by Super class method.

It means, if Super class method throws object of **NullPointerException** class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of **Exception** class (parent of NullPointerException class).

===============================================================================

To call the **run ()** method, **start ()** method is used. On calling start (), a new stack is provided to the thread and run () method is called to introduce the new thread into the program.

#### What if we call run () method directly without using start () method?

In above program if we directly call **run ()** method, without using **start ()** method,

public static void main (String args[] )

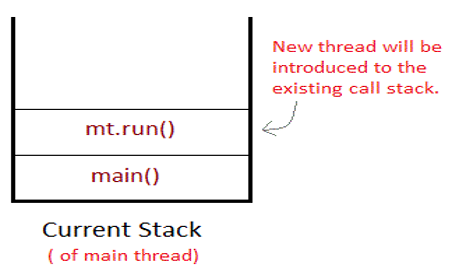
{

MyThread mt = new MyThread ();

mt.run ();

}

Doing so, the thread won't be allocated a new call stack, and it will start running in the current call stack, that is the call stack of the **main** thread. Hence Multithreading won't be there.



#### Can we Start a thread twice?

No, a thread cannot be started twice. If you try to do so, **IllegalThreadStateException** will be thrown.

public static void main (String args [])

{

MyThread mt = new MyThread ();

mt.start();

mt.start(); //Exception thrown

}

### **Joining threads**

Sometimes one thread needs to know when other thread is terminating. In java, **isAlive ()** and **join()** are two different methods that are used to check whether a thread has finished its execution or not.

The **isAlive ()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

**Join ()**

This method waits until the thread on which it is called terminates.

final void join () throws InterruptedException

Using **join ()** method, we tell our thread to wait until the specified thread completes its execution.

final void join (long milliseconds) throws InterruptedException

#### Example of thread without join () method

public class MyThread extends Thread

{

public void run ()

{

System.out.println("r1 ");

try {

Thread.sleep(500);

}

catch (InterruptedException ie) {}

System.out.println("r2 ");

}

public static void main(String [] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t2.start();

}

}

O/P:

r1

r1

r2

r2

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 ms.

At the same time Thread t2 will start its process and print "r1" on console and then go into sleep for 500 ms. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So, you will get output like r1 r1 r2 r2

#### Example of thread with join () method

public class MyThread extends Thread

{

public void run()

{

System.out.println("r1 ");

try {

Thread.sleep(500);

}catch(InterruptedException ie){ }

System.out.println("r2 ");

}

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start ();

try{

t1.join(); //Waiting for t1 to finish

}catch(InterruptedException ie){}

t2.start();

}

}

r1

r2

r1

r2

In this above program join () method on thread t1 ensures that t1 finishes it process before thread t2 starts.

#### Why we use Synchronization?

If we do not use synchronization and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, suppose we have two different threads **T1** and **T2**, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be **locked** (LOCK mode), and no other thread will be able to access or modify it until T1 returns.

#### Difference between synchronized keyword and synchronized block

When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that **no other thread can use any synchronized method** until the current thread, which has invoked its synchronized method, has finished its execution.

synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

#### Which is more preferred - Synchronized method or Synchronized block?

In Java, synchronized keyword causes a performance cost. A synchronized method in Java is very slow and can degrade performance. So, we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

### **Interthread Communication**

The **wait ()**, **notify ()**, and **notifyAll ()** methods of Object class are used for this purpose. These methods are implemented as **final** methods in Object, so that all classes have them. All the three methods can be called only from within a **synchronized** context.

* **wait ()** tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
* **notify ()** wakes up a thread that called wait () on same object.
* **notifyAll ()** wakes up all the thread that called wait () on same object.

#### Difference between wait () and sleep ()

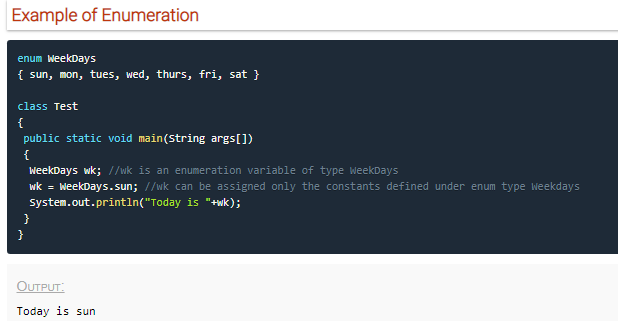
|  |  |
| --- | --- |
| **wait ()** | **sleep ()** |
| called from synchronized block | no such requirement |
| monitor is released | monitor is not released |
| gets awake when notify () or notifyAll () method is called. | does not get awake when notify () or notifyAll () method is called |
| not a static method | static method |
| wait () is generally used on condition | sleep () method is simply used to put your thread on sleep. |

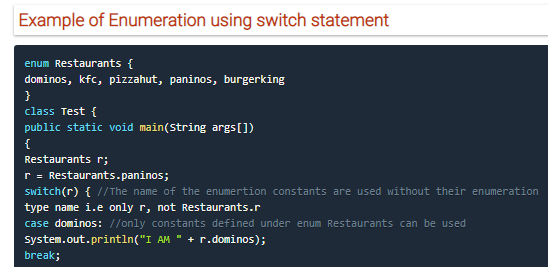
### **Enumerations**

**Enumeration** means a list of named constant. In Java, enumeration defines a class type. An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword.

Each enumeration constant is *public*, *static* and *final* by default. Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**.

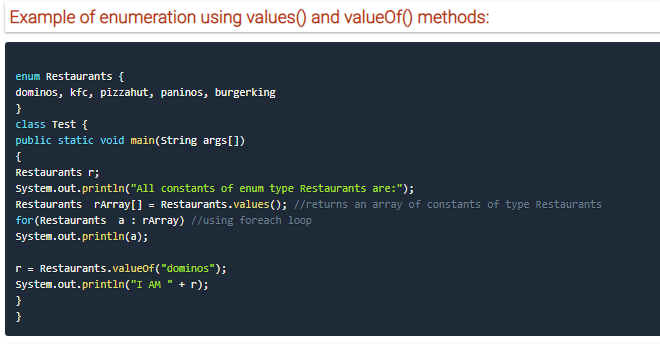
**Ex:**





**Points to remember about Enumerations**

1. Enumerations are of class type and have all the capabilities that a Java class has.
2. Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.
3. Enumerations are not instantiated using **new** keyword.
4. All Enumerations by default inherit **java.lang.Enum** class.

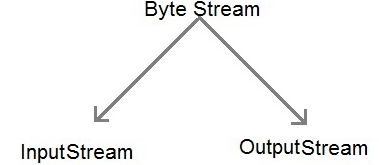


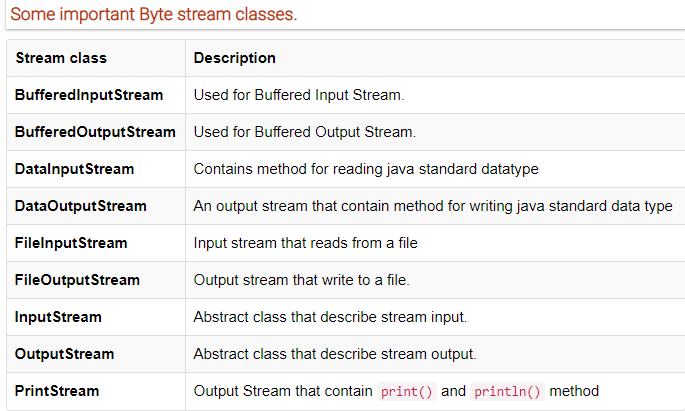


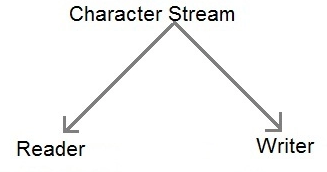
### **IO Stream**

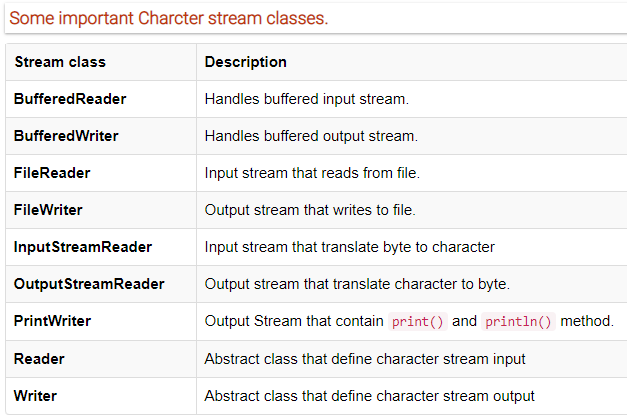
Java performs I/O through **Streams.**

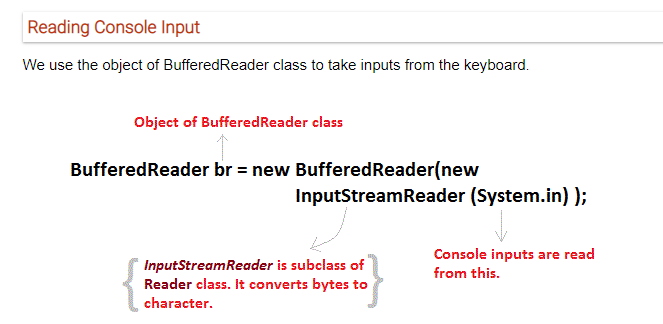
Java encapsulates Stream under **java.io** package. Java defines two types of stream.











### **Serialization and Deserialization in Java**

**Serialization** is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called **deserialization**.

**Serializable** is a **marker interface** that adds serializable behavior to the class implementing it.

Java provides **Serializable** API encapsulated under java.io package for serializing and deserializing objects which include,

* java.io.serializable
* java.io.Externalizable
* ObjectInputStream
* and ObjectOutputStream

while serializing if you do not want any field to be part of object state then declare it either **static** or **transient** based on your need and it will not be included during java serialization process.

Static members are never serialized because they are connected to class not object of class.

**Marker interface**

Marker Interface is a special interface in Java without any field and method. Marker interface is used to inform compiler that the class implementing it has some special behavior or meaning. Some example of Marker interface is,

* java.io.serializable
* java.lang.Cloneable
* java.rmi.Remote
* java.util.RandomAccess

All these interfaces do not have any method and field. They only add special behavior to the classes implementing them.

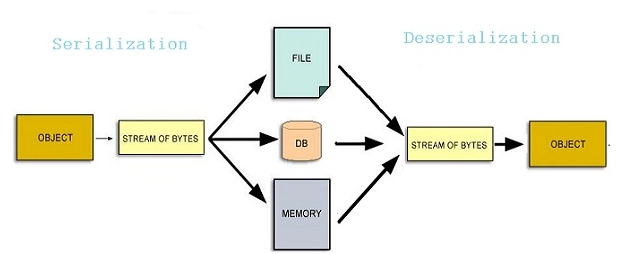
#### Signature of writeObject() and readObject()

writeObject() method of ObjectOutputStream class serializes an object and send it to the output stream.

public final void writeObject(object x) throws IOException

readObject() method of ObjectInputStream class references object out of stream and deserialize it.

public final Object readObject() throws IOException,ClassNotFoundException



#### Serializing an Object

import java.io.\*;

class studentinfo implements Serializable

{

String name;

int rid;

static String contact;

studentinfo(string n, int r, string c)

{

this.name = n;

this.rid = r;

this.contact = c;

}

}

class Test

{

public static void main(String[] args)

{

try

{

Studentinfo si = new studentinfo("Abhi", 104, "110044");

FileOutputStream fos = new FileOutputStream("student.ser");

Objectoutputstream oos = new ObjectOutputStream(fos);

oos.writeObject(si);

oos.close();

fos.close();

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

Object of Studentinfo class is serialized using writeObject() method and written to student.ser file.

#### Deserialization of Object

import java.io.\* ;

class DeserializationTest

{

public static void main(String[] args)

{

studentinfo si=null ;

try

{

FileInputStream fis = new FileInputStream("student.ser");

ObjectOutputStream ois = new ObjectOutputStream(fis);

si = (studentinfo)ois.readObject();

}

catch (Exception e)

{

e.printStackTrace(); }

System.out.println(si.name);

System.out. println(si.rid);

System.out.println(si.contact);

}

}

### **Generics**

#### Example of Generic class

class Gen <T> //<> brackets indicates that the class is of generic type

{

T ob; //an object of type T is declared

Gen(T o) //constructor

{

ob = o;

}

public T getOb()

{

return ob;

}

}

class Test

{

public static void main (String[] args)

{

Gen < Integer> iob = new Gen<>(100); //instance of Integer type Gen Class.

int x = iob.getOb();

System.out.println(x);

Gen < String> sob = new Gen<>("Hello"); //instance of String type Gen Class.

String str = sob.getOb();

System.out.println(str);

}

}

#### Generics Work Only with Objects

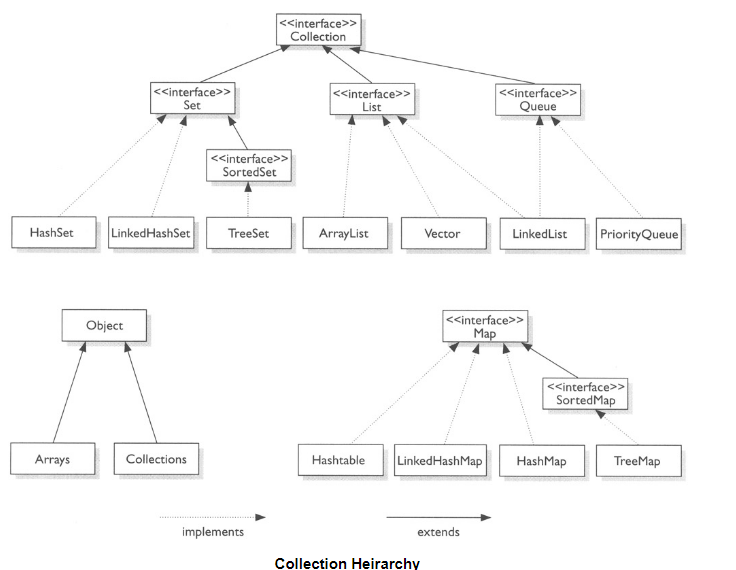
Gen<int> iOb = new Gen<int>(07); //Error, can't use primitive type

#### An array of Generic type cannot be created

T a []; **//this is allowed**

T a [] = new T [10]; **//this is not allowed**

### Collection Framework



#### Difference between ArrayList and Linked List

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| The ArrayList class creates the list which is internally stored in a **dynamic array** that grows or shrinks in size as the elements are added or deleted from it. | LinkedList also creates the list which is internally stored in a **DoublyLinked** List. |
| ArrayList allows **random access** to the elements in the list as it operates on an **index-based** data structure. | LinkedList does not allow random access as it does not have indexes to access elements directly, it must traverse the list to retrieve or access an element from the list. |
| ArrayList extends **AbstractList** | LinkedList extends **AbstractSequentialList**. |
| AbstractList implements **List** interface, thus it can behave as a list only | LinkedList implements **List**, **Deque** and **Queue** interface, thus it can behave as a Queue and List both. |
| Access to elements is faster in ArrayList as random access is also possible | Access to LinkedList elements is slower as it follows sequential access only. |
| Manipulation of elements is **slower** in ArrayList | **Faster** in LinkedList. |

There are two differences between **Vector** and **ArrayList**. First, Vector is synchronized while ArrayList is not, and Second, it contains many legacy methods that are not part of the Collections Framework.

### **HashSet class**

1. HashSet extends **AbstractSet** class and implements the **Set** interface.
2. It creates a collection that uses hash table for storage. A hash table stores information by using a mechanism called **hashing**.
3. In hashing, the informational content of a key is used to determine a unique value, called its **hash** **code**. The hash code is then used as the index at which the data associated with the key is stored.
4. HashSet does **not maintain any order** of elements.
5. HashSet contains only **unique** elements.

### **LinkedHashSet Class**

1. LinkedHashSet class extends **HashSet** class
2. LinkedHashSet stores elements in the order in which elements are inserted i.e it maintains the insertion order.

### **TreeSet Class**

1. It extends **AbstractSet** class and implements the **NavigableSet** interface.
2. It stores the elements in **ascending** order.
3. It uses a Tree structure to store elements.
4. It contains **unique** **elements** only like HashSet.
5. It's access and **retrieval** times are quite **fast**.

### **Accessing a Collection**

#### Accessing elements using Iterator

Iterator Interface is used to traverse a list in **forward** direction, enabling you to remove or modify the elements of the collection

#### Accessing elements using ListIterator

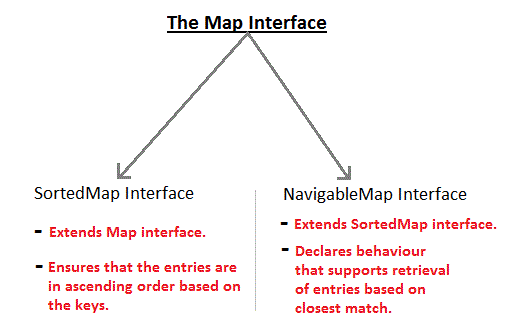
ListIterator Interface is used to traverse a list in both **forward** and **backward** direction. It is available to only those collections that implements the List Interface.

### **Map Interface**

|  |  |
| --- | --- |
| **Interface** | **Description** |
| **Map** | Maps unique key to value. |
| **Map.Entry** | Describe an element in key and value pair in a map. Entry is sub interface of Map. |
| **NavigableMap** | Extends SortedMap to handle the retrienal of entries based on closest match searches |
| **SortedMap** | Extends Map so that key are maintained in an ascending order. |

**Difference between HashMap and Hashtable**

|  |  |
| --- | --- |
| **Hashtable** | **HashMap** |
| Hashtable class is synchronized. | HashMap is not synchronized. |
| Because of Thread-safe, Hashtable is slower than HashMap | HashMap works faster. |
| Neither **key** nor **values** can be null | Both **key** and **values** can be null |
| Order of table remain constant over time. | does not guarantee that order of map will remain constant over time. |



#### HashMap class

1. HashMap class extends **AbstractMap** and implements **Map** interface.
2. It uses a **hashtable** to store the map. This allows the execution time of get() and put() to remain same.

#### TreeMap class

1. TreeMap class extends **AbstractMap** and implements **NavigableMap** interface.
2. It creates Map, stored in a tree structure.
3. A **TreeMap** provides an efficient means of storing key/value pair in efficient order.
4. It provides key/value pairs in sorted order and allows rapid retrieval.

#### LinkedHashMap class

1. **LinkedHashMap** extends **HashMap** class.

### **Legacy Classes**

Legacy classes defined by **java.util** package

* Dictionary
* HashTable
* Properties
* Stack
* Vector

There is only one legacy interface called **Enumeration**

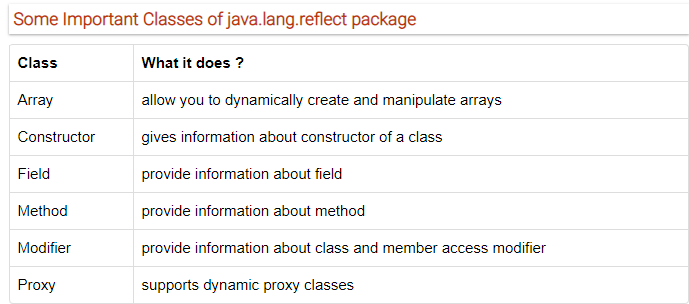
**Note\*:** All the legacy classes are synchronized

### **Reflection API**

Using reflection at the runtime you can determine what method, field, constructor or modifiers a class supports.

#### What is reflect package?

java.lang.reflect package encapsulates several important interfaces and classes. These classes and interface define methods which are used for reflection.



### **java.lang.Class class**

Class is a final class in **java.lang** package which extends **Object** class. Instance of this class represents classes and interfaces in a running Java application. It is used to analyze and change dynamic behaviour of a class at runtime.

Methods of java.lang.Class class:

#### forName ()

This method takes fully qualified name of classes or interface as its argument and returns instance of the class associated with it.

#### Example using forName () method

class Student {}

class Test

{

public static void main(String args[])

{

Class c = Class.forName("Student");

System.out.println(c.getName());

}

}

O/P: Student

#### getConstructors() and getDeclaredConstructors()

#### Example using getConstructors() and getDeclaredConstructors() method

import java.lang.reflect.\*;

class Student

{

public Student(){}

public Student(String name){}

}

class Test

{

public static void main(String args[])

{

try

{

Class c = Class.forName("Student");

Constructor< Student>[] ct = c.getConstructors();

for(int i=0; i< ct.length; i++)

{ System.out.println(ct[i]); }

Constructor< Student>[] cdt = c.getDeclaredConstructors();

for(int i=0;i< cdt.length;i++)

{ System.out.println(cdt[i]);}

}

catch(Exception e)

{ e.printStackTrace();}

}

}

#### 

#### getMethods() and getDeclaredMethods()

getMethods() method returns array of Method object that reflect all the public method of invoking

object. getDeclaredMethods() returns only the declared methods of the invoking class object

#### 

#### getFields() and getDeclaredFields()

getFields() returns an array containing Field objects reflecting all the accessible public members of the class or interface represented by this Class object. getDeclaredFields() returns array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.

#### 

### **Introduction to JDBC**

#### Java Database Connectivity(JDBC) is an Application Programming Interface(API) used to connect Java application with Database.

#### 

#### 4 Types of JDBC Drivers are present:

**1**. **Type-1 Driver** or **JDBC-ODBC bridge**

#### 

1. **Type-2 Driver or Native API Partly Java Driver**

#### 

1. **Type-3 Driver or Network Protocol Driver**

#### 

#### 4. Type-4 Driver or Thin Driver

#### 

#### JDBC 4.0 API is mainly divided into two packages

#### 

#### 

#### Steps to connect a Java Application to Database

#### 

#### Register the Driver

Class.forName() is used to load the driver class explicitly.

**Example to register with JDBC-ODBC Driver**

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

#### Create a Connection

getConnection() method of **DriverManager** class is used to create a connection.

Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","username","password");

#### Create SQL Statement

createStatement() method is invoked on current **Connection** object to create a SQL Statement

Statement s=con.createStatement();

#### Execute SQL Statement

executeQuery() method of **Statement** interface is used to execute SQL statements.

ResultSet rs=s.executeQuery("select \* from user");

while(rs.next())

{

System.out.println(rs.getString(1)+" "+rs.getString(2));

}

#### Closing the connection

After executing SQL statement, you need to close the connection and release the session. The close () method of **Connection** interface is used to close the connection.

con.close();

### **Connecting to Oracle Database using Thin Driver**

To connect a Java application with Oracle database using Thin Driver. You need to follow the following steps

1. **Load Driver Class:** The Driver Class for oracle database is **oracle.jdbc.driver.OracleDriver** and Class.forName("oracle.jdbc.driver.OracleDriver") method is used to load the driver class for Oracle database.
2. **Create Connection:** For creating a connection you will need a Connection URL. The Connection URL for Oracle is

#### 

#### The JDBC Statement, CallableStatement and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

#### Statement: Use the for general-purpose access to database. Useful when we are using static SQL statements at runtime. The Statement interface cannot accept parameters.

#### PreparedStatement: Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.

#### CallableStatement: Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

#### Execute Methods:

**boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

**int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

**ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

#### 

# **Java ResultSetMetaData Interface**

#### If you must get metadata of a table like total number of column, column name, column type etc., ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

|  |  |
| --- | --- |
| Method | Description |
| public int getColumnCount () throws SQLException | It returns the total number of columns in the ResultSet object. |
| public String getColumnName (int index) throws SQLException | It returns the column name of the specified column index. |
| public String getColumnTypeName (int index) throws SQLException | It returns the column type name for the specified index |
| public String getTableName (int index) throws SQLException | It returns the table name for the specified column index. |

### **Example of ResultSetMetaData interface:**

**import** java.sql.\*;

**class** Rsmd{

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

**try**{

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

 PreparedStatement ps=con.prepareStatement("select \* from emp");

ResultSet rs=ps.executeQuery();

ResultSetMetaData rsmd=rs.getMetaData();

 System.out.println("Total columns: "+rsmd.getColumnCount());

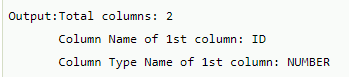
System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));

System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));

 con.close();

}**catch**(Exception e){ System.out.println(e);}

}  }



### **Simple Example of DatabaseMetaData interface :**

#### 

#### 