**Source File Declaration Rules:**

●One public class per source code file (we cannot define more source class in that)

●If there is a public class in the file, the name of the file must match the public class name

* There might not be any public class at all. Code will compile fine. However, to invoke the jvm with java, we have to follow a little different approach:

**Consider the following example:**

**Source File name:**

class A

{

}

class B extends A

{

public static void main(String []args)

{

A myA=new B();

m2(myA);

}

public static void m2(A a)

{

if(a instanceof B)

{

((B)a).dostuff();

}

}

public void dostuff()

{

System.out.println("a refers to B");

}

}

Now, define it in the file named Example2.java

When you compile the file, two class file (with .class extension) will be generated.

Now, when you want to run the program and you invoke jvm using java command, you have to submit the class name which contains definition of main. Hence,

javac Example2.java  
java B

●If the class is the part of a package, the package statement should be the first line of the source code before any import statements.’

●If there are import statements, they must go between the package name (if there is one) and class declaration

**Java Program Compilation And Execution:**

**Using The javac and java commands:**

**Compiling With Javac:**

The javac command is used to invoke java’s compiler. You can specify many options when running javac.

And what are those options like:

For instances, there are options to generate debugging information or compiler warning.

javac [options] [source-files]

Some of the examples of javac command:

Javac -help

Javac -version foo.java Bar.java

The first invocation does not compile any files, but prints a summary of valid options.

The second invocation passes the compiler an option, (-version, which prints the version of the compiler they are using) and passes the compiler two java files to compile, (foo.java and bar.java). Whenever, you are specifying multiple options, they must be separated by spaces.

**Launching Applications With Java: java command:**

The java command is used to invoke the java virtual machine.

**Import:**

In Java, the import statement is used to bring certain classes or the entire packages, into visibility. As soon as imported, a class can be referred to directly by using only its name. (Whether the package or class is library package or class, or user defined package or class, does not matter)

The import statement is a convenience to the programmer and is not technically needed to write complete Java program. If you are going to refer to some few dozen classes into your application, the import statement will save a lot of time and typing also.

**Static Import:**for instance, consider the following sample program:

**import static java.lang.System.out;**

**import static java.lang.Integer.\*;**

**public class TestStaticImport**

**{**

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

**{**

**out.println(MAX\_VALUE);**

**out.println(toHexString(42));**

**}**

**}**

Let’s look what’s happening in the code that’s using the static import.

* Even through the feature is static import, the syntax must be import static followed by the fully qualified name of the static member you want to import. Or, you could use wildcard. (**like, import static java.lang.Integer.\*; here, \* is the wildcard)**
* What does static import actually mean? For instance, **import static java.lang.Integer.\*;** it says I want to do static imports of all the static members.
* Now, we are fully seeing the benefits of static import features. First, it imports all the static members, second, We did not have to type System in System.out.println. Second thing, we don’t need to type the Integer in Integer.MAX\_VALUE. So, in this line, we were able to use a short cut for a static\_method and a constant.

(However, in the last case, where we don’t have to import the integer in Integer.MAX\_VALUE, watch out for the ambiguously named static members. For instance, if your program does a static import for both the classes Integer and Long, referring to the MAX\_VALUE will cause a compiler error. Since, for both classes static definition of MAX\_VALUE is present and compiler does not know which MAX\_VALUE you are referring.)

**Wildcard Concept In import:**

As you have seen, when using import and import static statements, sometimes you can use the wildcard character \* to do the simple searching (for a function or constant) for you. **(with the use of \* character, you can search through a package or within a class). you can say this:**

**import java.util.\*; //ok, to search the whole java.util packages**

In a similar vein, if you want to search the java.lang.Integer class for static members, you can say that:

**import static java.lang.integer.\*;**

But, you cannot create broader searches. For instance, you cannot use an import to search through the entire java API;

**import java.\*;**

**Data Types In Java:**

**Primitive Data Types:**



Note, In java, char is of 2 bytes.

**User Defined Data Type:**Probably enum is one of the user defined datatype. As of java 5, java lets you restrict a variable to have one of the predefined values. In other words, one value from an enumerated list.

Using enum can help in reducing the bugs in your code.

For instance, in your coffee shop application, you might want to restrict your coffee size size selections to BIG, HUGE, and OVERWHELMING. If you let and order for a **LARGE** or **GRANDE** slip in, it might cause an error.

An enum can be defined as the following:

**enum CoffeeSize{BIG, HUGE, OVERWHEELMING};**

It’s not required that enum constants be all in caps, but borrowing from the Oracle code conventions, **that constants are named in caps**, it’s a good idea.

**Now, enum could be declared out side of a class, it could be declared within a class as a class member, or enum can be declared as their own separate class.**

**An Example: (when enum is declared outside of any class)**

enum CoffeeSize{BIG, HUGE, OVERWHELMING}

//no semicolon at the end

//this cannot be private or protected

class Coffee

{

CoffeeSize size;

}

public class CoffeeTest1

{

public static void main(String args[])

{

Coffee drink=new Coffee();

drink.size=CoffeeSize.BIG;

System.out.println("The drink size is: "+drink.size);

}

}

**Note the following things:**

1. Both class Coffee and CoffeeTest1 is in the same package. No access specifier is specified to the class Coffee as well as it’s members. Hence, the scope, the package default. For class Coffee as well as the variable size in it. Hence,   
     
   drink.size=CoffeeSize.BIG can be accessed.
2. We have a enum outside the class’s scope.
3. The way in which we can access one of the newly defined values.
4. note that thing. Java language designers make it optional to put a semicolon at the end of the enum declaration. So, what gets created when you make an enum? The most important thing to remember that an enum is not string or int. Each of the enumerated CoffeeSize typesare actually an instance of CoffeeSize. Think of an enum as a kind of class that looks something like this (not exactly though)

**class CoffeeSize**

**{**

**public static final CoffeeSize BIG=new CoffeeSize(“BIG”,0);**

**public static final CoffeeSize HUGE=new CoffeeSize(“HUGE”,1);**

**public static final CoffeeSize OVERWHELMING=new CoffeeSize(“OVERWHELMING”,”2”);**

**CoffeeSize(String enumName, int index)**

**{**

**//stuff here**

**}**

**}**

**How can you know the exact structure:**

**Consider the example:**

enum CoffeeSize{BIG, HUGE, OVERWHELMING}

//no semicolon at the end

//this cannot be private or protected

class Coffee

{

CoffeeSize size;

}

public class CoffeeTest1

{

public static void main(String args[])

{

Coffee drink=new Coffee();

drink.size=CoffeeSize.BIG;

System.out.println("The drink size is: "+drink.size);

}

}

First, compile it : javac CoffeeTest1.java  
  
After that disassemble the enum definition using

**javap CoffeeSize**

It will generate the following thing:

final class CoffeeSize extends java.lang.Enum<CoffeeSize> {

public static final CoffeeSize BIG;

public static final CoffeeSize HUGE;

public static final CoffeeSize OVERWHELMING;

public static CoffeeSize[] values();

public static CoffeeSize valueOf(java.lang.String);

static {};

}

The disassemble shows that that each field of an enum is an instance of the Constants enum class. (Further analysis with javap will reveal that each field is initialized by creating a new object by calling the new Constants(String) constructor in the static initialization block.)

Therefore, we can tell that each enum field that we create will be at least as much as the overhead of creating an object in the JVM.

**Declaring Constructors, Methods, Variables In An Enum:**

Because, enum is a special kind of class, you can do more than just list the enumerated constant values. You can add constructors, instance variables, methods and something really strange known as a constant specific class body. To understand, why you might need more in your enum, think about the particular scenario: imagine you want to know the actual size, in ounces, that map to the three CoffeeSize constants. Now, you could make some kind of lookup table using some other data structures. But that will be a poor design and hard to maintain. The simplest way to treat your enum values as objects, each of which can have its own instance variables and own values.

enum CoffeeSize

{

BIG(8), HUGE(10), OVERWHELMING(16);

private int ounces;

CoffeeSize(int ounces)

{

this.ounces=ounces;

}

public int getOunces()

{

return ounces;

}

}

public class Coffee

{

CoffeeSize size;

public static void main(String[] args)

{

Coffee drink1=new Coffee();

drink1.size=CoffeeSize.BIG;

System.out.println("In "+drink1.size+" we get "+drink1.size.getOunces());

}

}

Which produces: In BIG we get 8

There are some points to know:

●You can never invoke an enum constructor directly. The enum constructor is invoked automatically, with the arguments you defined after the constant value.

●You can define more than one argument to the constructor, and you can overload the enum constructors. Just as you overload a normal class constructor.

**Variable Declarations:**there are two types of variables in java.   
  
**Primitives:** a **primitive** can be one of the eight types. Char, boolean, short, int, long, double or float. Once, a primitive has been declared, its primitive type can never be changed. Although in most cases, its value can be changed.

Now, java char has 2 bytes

**Reference Variables:** a reference variable is used to refer to an object. A reference variable is declared to be a specific type and that type can never be changed. A reference variable can be used to refer to any other objects of the declared type or of a subtype of the declared type.

**Pass By Reference Concept In Java:  
  
If Java uses the pass-by reference, why won't a swap function work?**  
A: Java does manipulate objects by reference, and all object variables are references. However, Java doesn't pass method arguments by reference; it passes them by value.  
  
Take the badSwap() method for example:  
  
public void badSwap(int var1, int var2)  
{  
  int temp = var1;  
  var1 = var2;  
  var2 = temp;  
}  
When badSwap() returns, the variables passed as arguments will still hold their original values. The method will also fail if we change the arguments type from int to Object, since Java passes object references by value as well.

so , what does that mean?

**import java.util.Vector;**

**public class Test**

**{**

**public static void addInContainer(Vector<Integer> container)**

**{**

**container.add(20);**

**container.add(30);**

**}**

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

**{**

**Vector<Integer> container=new Vector<>();**

**container.add(10);**

**//Now, print the values**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**addInContainer(container);**

**System.out.println("After calling addInContainer");**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**}**

**}**

**This will print:**

10   
After calling addInContainer  
10 20 30

**import java.util.Vector;**

**public class Test**

**{**

**public void swap(Vector<Integer> container)**

**{**

**Vector<Integer> container2=new Vector<Integer>();**

**container=container2;**

**}**

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

**{**

**Vector<Integer> container=new Vector<>();**

**container.add(10);**

**container.add(20);**

**container.add(30);**

**//Now, print the values**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**System.out.println("After swap function is called");**

**for(int i=0;i<container.size();i++)**

**{**

**System.out.print(container.elementAt(i)+" ");**

**}**

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

**}**

**}**

**This will print:**

10 20 30   
After swap function is called  
10 20 30

**So, what do you learn:**

1. Java does not pass primitive data types as reference as function arguments
2. For Objects, java passes references in a copy by value approach.

**Packages In Java:**

**There are two kinds of packages:**

1) User defined package: The package we create is called user-defined package.

2) Built-in package: The already defined package like java.io.\*, java.lang.\* etc are known as built-in packages.

**If a class is under a package, the statement package …; should be the first statement.**

**Consider the following example:**

package cert;

public class sludge

{

public void testit(){System.out.println(“Sludge”)};

}

In this class, **package cert;** should be the first statement. All imports come later.

**Sub packages in Java**

A package inside another package is known as sub package. For example If I create a package inside letmecalculate package then that will be called sub package.

Lets say I have created another package inside letmecalculate and the sub package name is multiply. So if I create a class in this subpackage it should have this package declaration in the beginning:

**package letmecalculate.multiply;**

**Note:** If you don’t declare a package for a class, it will be part of default package.

java is a package centric language, the developers assumed that for **good organization and for named scoping**, you would put all your classes into packages. And, this is right, Otherwise, consider the following situation. Three different programmers which are in the same company but working on different projects,define their own utilities class. Now, if those classes are not declared in any of the explicit package, and are in the class path, there is no way to tell JVM or compiler which one you are trying to reference.

Oracle suggests/recommends that developer use **reverse domain names appended with division and/or project names. For example, if your domain name is anonymous.com your package name should start with com.anonymous.**

**Class Declarations And Modifiers:**

Class modifiers are fall into two types:

**Access Modifiers. (public, private, protected)**

**Non access modifiers. (strictfp, final and abstract)**

**Access Modifier:**Now, though there is three access modifiers, public, private and protected, there are four levels of access controls. As the fourth one is default or package access when you don’t use any of the chosen access modifiers. (private, protected, public)

However, all the four access modifiers are for class variables and functions. For a class, there is two access modifiers. **Public and Default**

**What does that mean? A class has only two access specifiers, while a variable can have all the four access specifiers.**

**Public access modifier:**

When a method or variable member is declared public, it means all other classes, regardless of the package they belong to, can access the member. (Assuming that the class itself is visible)

**package cert;**

**public class sludge**

**{**

**public void testit(){System.out.println(“Sludge”)};**

**}**

**package book;**

**import cert.\*;  
class Goo**

**{**

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

**{**

**sludge o=new sludge();**

**O.testIt();**

**}**

**}**

As you can see, Goo and sludge are in different packages. However, Goo can invoke the method in sludge without problems, because, both the sludge class and it’s testIt() method are made public.

**But, if you make the following changes, it will not even compile:**

**Goo.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package book;

import cert.\*;

public class Goo

{

public static void main(String[] args)

{

Sludge o=new Sludge();

o.testIt();

}

}

**Cert.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package cert;

/\*\*

\*

\* @author Reve

\*/

public class Sludge

{

//it's access specifier is made package default

void testIt()

{

System.out.println("Sludge");

}

}

You cannot even compile it.

It will give the compilation error for following line in Goo.java

**o.testIt(); as testIt has default access.**

**Now, if you change the class definition to the following:**

**Goo.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package book;

import cert.\*;

public class Goo

{

public static void main(String[] args)

{

Sludge o=new Sludge();

o.testIt();

}

}

**Cert.java**

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package cert;

/\*\*

\*

\* @author Reve

\*/

class Sludge

{

//it's access specifier is made package default

public void testIt()

{

System.out.println("Sludge");

}

}

It will give compilation error, too. But, in this line:

**Sludge o=new Sludge();**

Note: main must be contained under a public class. (Thus, every java project **(does not matter if it consists of even one file should contain at least one public class)**

**Private access modifiers:**

Members marked private cannot be accessed by code in other class other than the class in which the private member is declared.   
  
  
**Now, Note that,** a private method of a super class cannot be overridden by a subclass since, it is not inheriting it.

**Now, note that,** private access modifier cannot be applied to a class

**Protected Access modifiers:**

The protected and default access control levels are almost identical. But, with one critical difference. A default member may be accessed only if the class accessing the member belongs to thee same package. Whereas, a protected member can be accessed (through inheritance) by a subclass if the subclass is in a different package.

**Any class could only see the protected members through inheritance.**

**Non Access Modifiers Of A Class:**

strictfp, final and abstract. (what is native? Native is a modifier used on a function name. But it finally uses the native interface to reuse the functions defined in other languages)

**Final Class:**when used in a class declaration, the final keyword means the class cannot be sub classed. In other words, no other class can ever extend a final class. And trying to do so will generate **Compilation error. In other words, no other class can ever extend.** You should make a final class only if you need an absolute guarantee that none of the methods in that class will ever be overridden.

**Many classes in java core libraries are final. Like, String class.** Imagine the havoc if you could not guarantee how a string object would work on any given system your application is running on.

**However, in practice, we will almost never make a final class.** A final class obliterates a key benefit of OO -extensibility. So, unless if you have a serious safety or security issue, you should not do this.

**Final Functions:**

prevents a method from being overridden in a subclass.

**Final Arguments:**

Final keyword can also be used in case of an argument. The concept is similar to const argument in c++. **A new value cannot be assigned to the variable. If that is passed ass a final argument.**

**Abstract Class:**

An abstract class can never be instantiated. So, it’s purpose is it has to be extended.

**So, conceptually where it is useful?**

Imagine you have a class car that has generic methods common to all vehicles. But, you don’t want anyone to actually create a generic, abstract class object.

Or, the bank account example.

**abstract class Car**

**{**

**private double Price;**

**private String model;**

**private String year;**

**private abstract void goFast();**

**private abstract void goUpHill();**

**private abstract void impressNeighbours();**

**}**

**Some points about abstract class:**

* Even a single method is abstract in a normal class, that class has to be defined as abstract.
* However, you can have non abstract methods in a abstract class. For example, you might have methods that should not change from Car Type to Car Type such as getColor() or setPrice(). **By putting non-abstract methods in a abstract class, you give all concrete subclasses inherited method implementation.**
* **An interface cannot be abstract**
* **A variable cannot be abstract.**

**Strictfp modifier:**

**strictfp** is a keyword in java used for restricting floating-point calculations and ensuring same result on every platform while performing operations in the floating-point variable.  
Floating point calculations are platform dependent i.e. different output(floating-point values) is achieved when a class file is run on different platforms(16/32/64 bit processors). To solve this types of issue, strictfp keyword was introduced in JDK 1.2 version by following [IEEE 754](https://en.wikipedia.org/wiki/IEEE_floating_point) standards for floating-point calculations.

**Important points:**

* strictfp modifier is used with classes, interfaces and methods only.

**strictfp class Test**

**{**

**// all concrete methods here are**

**// implicitly strictfp.**

**}**

**strictfp interface Test**

**{**

**// all methods here becomes implicitly**

**// strictfp when used during inheritance.**

**}**

**class Car**

**{**

**// strictfp applied on a concrete method**

**strictfp void calculateSpeed(){}**

**}**

* strictfp modifier cannot be used with variables.
* When a class or an interface is declared with strictfp modifier, then all methods declared in the class/interface, and all nested types declared in the class, are implicitly strictfp.
* strictfp cannot be used with abstract methods. However, it can be used with abstract classes/interfaces. **(since, only concrete methods can be strictfp)**
* Since methods of an interface are implicitly abstract, strictfp cannot be used with any method inside an interface. **(Because, strictfp can only be used with concrete functions)**

**Example 1:**

**strictfp interface Test**

**{**

**double sum();**

**double mul();**

**}**

It is allowed

**Example 2:  
  
 strictfp interface Test**

**{**

**double sum();**

**strictfp double mul(); // compile-time error here**

**}**

It is not.

**Example 3:**

//Java program to illustrate strictfp modifier

**public class Test**

**{**

**// calculating sum using strictfp modifier**

**public strictfp double sum()**

**{**

**double num1 = 10e+10;**

**double num2 = 6e+08;**

**return (num1+num2);**

**}**

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

**{**

**Test t = new Test();**

**System.out.println(t.sum());**

**}**

**}**

**User Interfaces:**When you create an interface, you are defining what a class can do without saying how the class will do it.

Any class type that implements the interface must write code for all methods.

**Difference between interface and abstract class:**

Now, abstract class-inheritance-is a relationship, Whereas, modular kitchen is implemented by normal kitchen

However, except the theoretical part, technically consider interface as a 100% abstract class

**Some Points For Interfaces:**

* All interface methods are implicitly public and abstract. In other words, you do not need to type public or abstract modifiers in the method declaration, but the method will always be public and abstract.
* All variables defined in an interface must be public, static and final. **(and these modifiers are not implicit. You have to mention it).** In other words, **interfaces can only have constants, not instance variables.**
* However, unlike, the variables, which should be **public, static and final,** interfaces method should not be static.
* Because, interface methods are abstract, they cannot be marked as final.
* **An interface can extend one or more other interfaces.**
* An interface cannot extend anything but another interface.
* **An interface cannot implements another interface or class.**
* An interface must be declared with the keyword interface

Like the following:  
 **public interface bouncable**

**{**

**void bounce();**

**void setbouncefactor(int bf);**

**}**

* Because, interface methods are abstract (implicitly), we cannot use **final, strictfp or native modifiers with them.**Final methods cannot be abstract.

strictfp modifer must be applied on concrete functions

native keyword is used to declare a function which is defined elsewhere. It cannot have a combination with abstract.

* Interface types can be used polymorphically.
* **Further Note:** the following is a legal interface declaration:

**public abstract interface Rollable()**

**{**

**//the variables (which must be static, public and final)**

**//the methods (which are implicitly public and abstract)**

**}**

However, typing in the **abstract modifier** is considered redundant.

**Static KeyWord In Java:**static is a non-access modifier in Java which is applicable for the following:

blocks

variables

methods

nested classes

**Static Blocks:**

1. The code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class). For example, check output of following Java program.

**First Example:**

// filename: Main.java

**class Test {**

**static int i;**

**int j;**

**// start of static block**

**static {**

**i = 10;**

**System.out.println("static block called ");**

**}**

**// end of static block**

**}**

**class Main {**

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

**// Although we don't have an object of Test, static block is**

**// called because i is being accessed in following statement.**

**System.out.println(Test.i);**

**}**

**}**

**Output:**

static block called

1. Also, static blocks are executed before constructors. For example, check output of following Java program:

**Another Example:**

// filename: Main.java

**class Test {**

**static int i;**

**int j;**

**static {**

**i = 10;**

**System.out.println("static block called ");**

**}**

**Test(){**

**System.out.println("Constructor called");**

**}**

**}**

**class Main {**

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

**// Although we have two objects, static block is executed only once.**

**Test t1 = new Test();**

**Test t2 = new Test();**

**}**

**}**

**Output:**

static block called

Constructor called

Constructor called

**Multiple Static Blocks:**

They execute in the given order which means the first static block executes before second static block. That’s the reason, values initialized by first block are overwritten by second block.

**class JavaExample2{**

**static int num;**

**static String mystr;**

**//First Static block**

**static{**

**System.out.println("Static Block 1");**

**num = 68;**

**mystr = "Block1";**

**}**

**//Second static block**

**static{**

**System.out.println("Static Block 2");**

**num = 98;**

**mystr = "Block2";**

**}**

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

**{**

**System.out.println("Value of num: "+num);**

**System.out.println("Value of mystr: "+mystr);**

**}**

**}**

**Static blocks’ Major Use:**If a class has static members that require complex initialization, a static block is the tool to use. Suppose you need a static map of some kind (the purpose is irrelevant here). You can declare it in-line like this:

**public static final Map<String, String> initials = new HashMap<String, String>();**

However, if you want to populate it once, you can't do that with an in-line declaration. For that, you need a static block:

**public static final Map<String, String> initials = new HashMap<String, String>();**

**static**

**{**

**initials.put("AEN", "Alfred E. Newman");**

**// etc.**

**}**

**Note:** This is the way to populate a map in a fixed manner. Otherwise, it is needed to be populated whenever we create an instance

Don’t get confused in the concept. We could very much do the following:

import java.util.HashMap;

class FinalMapTest

{

public static final HashMap<String,Integer> NameRollMap=new HashMap<>();

public void populateMap()

{

NameRollMap.put("Sayak",1);

}

}

public class Trial

{

public static void main(String args[])

{

FinalMapTest mapTest=new FinalMapTest();

mapTest.populateMap();

}

}

However, if you want to be more protective:

**public static final Map<String, String> initials;**

**static**

**{**

**HashMap<String, String> map = new HashMap<String, String>()**

**map.put("AEN", "Alfred E. Newman");**

**// etc.**

**initials = Collections.unmodifiableMap(map);**

**}**

Now, unmodifiableMap returns an unmodifiable view of the map.

**Another use could be System.loadLibrary() to link a native DDL (Dynamically Linked library) dynamically.**

**Static Functions:**

When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

**Important points for static variables :-**

* We can create static variables at class-level only.   
    
  (What does that mean? It means in java,

Static variables cannot be created to locally of a function in java. Can be created in c++.

class Test {

public static void main(String args[]) {

System.out.println(fun());

}

static int fun()

{

static int x= 10; //Error: Static local variables are not allowed

return x--;

}

}

**This is not allowed in java.**

* static block and static variables are executed in order they are present in a program.

Below is the java program to demonstrate that static block and static variables are executed in order they are present in a program.

// java program to demonstrate execution

// of static blocks and variables

class Test

{

// static variable

static int a = m1();

// static block

static {

System.out.println("Inside static block");

}

// static method

static int m1() {

System.out.println("from m1");

return 20;

}

// static method(main !!)

public static void main(String[] args)

{

System.out.println("Value of a : "+a);

System.out.println("from main");

}

}

**Can Static Members Of A Class Be Accessed Through Non Static Instance Function?**

Yes.

**Static Methods:**

static methods are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the Object of that class.

public static void geek(String name)

{

// code to be executed....

}

// Must have static modifier in their declaration.

// Return type can be int, float, String or user defined data type.

**Memory Allocation:**

They are stored in Permanent Generation space of heap as they are associated to the class in which they reside not to the objects of that class. But their local variables and the passed argument(s) to them are stored in the stack. Since they belong to the class so they can be called to without creating the object of the class.

**Important Points:**

Static method(s) are associated to the class in which they reside i.e. they can be called even without creating an instance of the class i.e **ClassName.methodName(args).**

* They are designed with aim to be shared among all Objects created from the same class.
* Static methods can not be overridden. But can be overloaded since those are resolved using static binding by compiler at compile time.
* A non static function cannot be from a static function without using an object.
* This reference does not exist for static functions.

**Static nested classes :**

We can not declare top-level class with a static modifier, but can declare nested classes as static. Such type of classes are called Nested static classes.

**What are the differences between static and non-static nested classes?**

Following are major differences between static nested class and non-static nested class. Non-static nested class is also called Inner Class.

1) Nested static class doesn’t need reference of Outer class, but Non-static nested class or Inner class requires Outer class reference.

/\* Java program to demonstrate how to implement static and non-static

classes in a java program. \*/

class OuterClass{

private static String msg = "GeeksForGeeks";

// Static nested class

public static class NestedStaticClass{

// Only static members of Outer class is directly accessible in nested

// static class

public void printMessage() {

// Try making 'message' a non-static variable, there will be

// compiler error

System.out.println("Message from nested static class: " + msg);

}

}

// non-static nested class - also called Inner class

public class InnerClass{

// Both static and non-static members of Outer class are accessible in

// this Inner class

public void display(){

System.out.println("Message from non-static nested class: "+ msg);

}

}

}

class Main

{

// How to create instance of static and non static nested class?

public static void main(String args[]){

// create instance of nested Static class

OuterClass.NestedStaticClass printer = new OuterClass.NestedStaticClass();

// call non static method of nested static class

printer.printMessage();

// In order to create instance of Inner class we need an Outer class

// instance. Let us create Outer class instance for creating

// non-static nested class

OuterClass outer = new OuterClass();

OuterClass.InnerClass inner = outer.new InnerClass();

// calling non-static method of Inner class

inner.display();

// we can also combine above steps in one step to create instance of

// Inner class

OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();

// similarly we can now call Inner class method

innerObject.display();

}

}

Now, look at this:

**OuterClass.NestedStaticClass printer = new OuterClass.NestedStaticClass();**

**printer.printMessage();**

For instantiating static inner class, we need not a reference of Outerclass which has to be instantiated with OuterClass’s object.

2) Inner class(or non-static nested class) can access both static and non-static members of Outer class. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class.

**Programming Paradigm, OOP:**

**What is Programming Paradigm?**

Answer) Programming paradigm is a fundamental style of computer programming. It is a way of building the structures and functions around the program.

**Object Oriented Programming Paradigm:**

**Answer)**

OOP treats data as a critical element and does not allow data to flow freely around the system.

OOP ties data close to the function which operates on it

OOP allows decomposition of a function into a set of entities called objects(runtime instance of class) and builds data data around them

The data of an object can be accessed only by the function associated with the objects.

One object can communicate with other objects using function of that object.

**Describe four key concepts of object oriented programming.**

Answer) The four key concepts of object oriented programming is 1) Encapsulation 2) Abstraction 3) Inheritance 4) Polymorphism

**1) Encapsulation:** The wrapping up of data and methods into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world. Only those methods, which are wrapped in the class, can access it.

So, these methods (which are wrapped in the class) provides the object's data and the program. This insulation of the data from direct access by the program is called 'data hiding'. (Private data members are example of data encapsulation)

**2) Abstraction:** Abstraction refers to the act of representing essential features of a class and omit the unnecessary details about it. For instance, when we think about a car, we don't consider all the irrelevant details like how the break actually works or how the Bluetooth device is installed in the

car, how it provides all the cool features. We think it as a transport medium which could take us from one place to another place. That is abstraction.

Now, the difference between abstraction and encapsulation:

One is mechanism hiding and another is data hiding.

**3) Inheritance:** Inheritance is a way by which a newly defined class inherits attributes and behaviour of an existing class along with its own properties.

Using inheritance the hierarchical relationships are established.

Inheritance allows the re usability of an existing operations and extending the basic unit of the a class without creating it from the scratch.

**Some more points about Inheritance:**

Inheritance is a “is a” relationship (not a “has a” relationship)

Like: we can say “A four-wheeler is a car” (It's a real “is a” relationship)

But, we cannot say, “A steering wheel is a car” (It's a “has a” relationship)

(A has a relationship is used in composition. An example of composition: A structure with in a

structure as a member)

Inheritance promotes both. Polymorphism and code re usability.

**4) Polymorphism:** Polymorphism is sharing a common interface for multiple types but having different implementation for different types.

In OOP, polymorphism is a technique where objects of classes belonging to

the same hierarchical tree may posses interface bearing the same name but each having different behaviors.

It is the way of inheriting when useful, overriding when not useful.

It allows automatically do the current behaviour even if we are working with

many different forms

**Define class.**

Answer) Class is a static definition of new type as a collection of data and associated operations from which runtime instances called objects can be created.

**Define object.**

Answer) Object is runtime instance of a conceptual framework encapsulating typed data and typed operations that correspond to a real world entity or thing for the purpose of computer modeling.

**Constructors:  
  
Default Constructor:**

class NoteBook{

/\*This is default constructor. A constructor does

\* not have a return type and it's name

\* should exactly match with class name

\*/

NoteBook(){

System.out.println("Default constructor");

}

public void mymethod()

{

System.out.println("Void method of the class");

}

public static void main(String args[]){

/\* new keyword creates the object of the class

\* and invokes constructor to initialize object

\*/

NoteBook obj = new NoteBook();

obj.mymethod();

}

}

**Now, Some Notes About Constructor:**

* If you don’t provide a default constructor, compiler will include one.
* However, if you overload it and instead of providing a no-arg constructor/default constructor, you provide a parametric constructor, you can no more call the default constructor.

**Consider the following program:**

class Example{

Example(int i, int j){

System.out.print("parameterized constructor");

}

Example(int i, int j, int k){

System.out.print("parameterized constructor");

}

public static void main(String args[]){

Example obj = new Example();

}

}

**This will generate the following output:**

Exception in thread "main" java.lang.Error: Unresolved compilation

problem: The constructor Example() is undefined

**Parametric Constructor:**class Example{

//Default constructor

Example(){

System.out.println("Default constructor");

}

/\* Parameterized constructor with

\* two integer arguments

\*/

Example(int i, int j){

System.out.println("constructor with Two parameters");

}

/\* Parameterized constructor with

\* three integer arguments

\*/

Example(int i, int j, int k){

System.out.println("constructor with Three parameters");

}

/\* Parameterized constructor with

\* two arguments, int and String

\*/

Example(int i, String name){

System.out.println("constructor with int and String param");

}

public static void main(String args[]){

//This will invoke default constructor

Example obj = new Example();

/\* This will invoke the constructor

\* with two int parameters

\*/

Example obj2 = new Example(12, 12);

/\* This will invoke the constructor

\* with three int parameters

\*/

Example obj3 = new Example(1, 2, 13);

/\* This will invoke the constructor

\* with int and String parameters

\*/

Example obj4 = new Example(1,"BeginnersBook");

}

**Copy Constructor:**

Now, like c++, java also supports copy constructor. But, unlike c++ compiler, java compiler does not include a default definition of of copy constructor.

**Example:**

class Complex {

private double re, im;

// A normal parametrized constructor

public Complex(double re, double im) {

this.re = re;

this.im = im;

}

// copy constructor

Complex(Complex c) {

System.out.println("Copy constructor called");

re = c.re;

im = c.im;

}

// Overriding the toString of Object class

@Override

public String toString() {

return "(" + re + " + " + im + "i)";

}

}

public class Main {

public static void main(String[] args) {

Complex c1 = new Complex(10, 15);

// Following involves a copy constructor call

Complex c2 = new Complex(c1);

// Note that following doesn't involve a copy constructor call as

// non-primitive variables are just references.

Complex c3 = c2;

System.out.println(c2); // toString() of c2 is called here

}

}

Now, you can twist the argument passing a little:

**Check the Following Example:**

class Complex {

private double re, im;

// A normal parametrized constructor

public Complex(double re, double im) {

this.re = re;

this.im = im;

}

// copy constructor

Complex(final Complex c) {

System.out.println("Copy constructor called");

re = c.re;

im = c.im;

}

// Overriding the toString of Object class

@Override

public String toString() {

return "(" + re + " + " + im + "i)";

}

}

public class Example2{

public static void main(String[] args) {

Complex c1 = new Complex(10, 15);

// Following involves a copy constructor call

Complex c2 = new Complex(c1);

// Note that following doesn't involve a copy constructor call as

// non-primitive variables are just references.

Complex c3 = c2;

System.out.println(c2); // toString() of c2 is called here

}

}

**See, you can make the reference final. That is like you can have definition. This is unlike c++ , again. Since, c++ provides a default definition of copy constructor, there’s a standard of it. If you are overloading copy constructor, you have to follow the standard. Here, the case is different. As java compiler does not provide default definition of copy constructor.**

**This Reference:**

**1. Using ‘this’ keyword to refer current class instance variables**

//Java code for using 'this' keyword to

//refer current class instance variables

class Test

{

int a;

int b;

// Parameterized constructor

Test(int a, int b)

{

this.a = a;

this.b = b;

}

void display()

{

//Displaying value of variables a and b

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

}

public static void main(String[] args)

{

Test object = new Test(10, 20);

object.display();

}

}

**2. Using this() to invoke current class constructor**

// Java code for using this() to

// invoke current class constructor

class Test

{

int a;

int b;

//Default constructor

Test()

{

this(10, 20);

System.out.println("Inside default constructor \n");

}

//Parameterized constructor

Test(int a, int b)

{

this.a = a;

this.b = b;

System.out.println("Inside parameterized constructor");

}

public static void main(String[] args)

{

Test object = new Test();

}

}

**Super Keyword In Java:**

1. **Use of super with variables:**

This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

**/\* Base class vehicle \*/**

**class Vehicle**

**{**

**int maxSpeed = 120;**

**}**

**/\* sub class Car extending vehicle \*/**

**class Car extends Vehicle**

**{**

**int maxSpeed = 180;**

**void display()**

**{**

**/\* print maxSpeed of base class (vehicle) \*/**

**System.out.println("Maximum Speed: " + super.maxSpeed);**

**}**

**}**

**/\* Driver program to test \*/**

**class Test**

**{**

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

**{**

**Car small = new Car();**

**small.display();**

**}**

**}**

**Output:**

Maximum Speed: 120

In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in sublcass using super keyword.

1. **Use of super with methods:**

This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

**/\* Base class Person \*/**

**class Person**

**{**

**void message()**

**{**

**System.out.println("This is person class");**

**}**

**}**

**/\* Subclass Student \*/**

**class Student extends Person**

**{**

**void message()**

**{**

**System.out.println("This is student class");**

**}**

**// Note that display() is only in Student class**

**void display()**

**{**

**// will invoke or call current class message() method**

**message();**

**// will invoke or call parent class message() method**

**super.message();**

**}**

**}**

**/\* Driver program to test \*/**

**class Test**

**{**

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

**{**

**Student s = new Student();**

**// calling display() of Student**

**s.display();**

**}**

**}**

**Output:**

This is student class

This is person class

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

1. **Use of super with constructors:**super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

**/\* superclass Person \*/**

**class Person**

**{**

**Person()**

**{**

**System.out.println("Person class Constructor");**

**}**

**}**

**/\* subclass Student extending the Person class \*/**

**class Student extends Person**

**{**

**Student()**

**{**

**// invoke or call parent class constructor**

**super();**

**System.out.println("Student class Constructor");**

**}**

**}**

**/\* Driver program to test\*/**

**class Test**

**{**

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

**{**

**Student s = new Student();**

**}**

**}**

**Output:**

Person class Constructor

Student class Constructor

In the above example we have called the superclass constructor using keyword ‘super’ via subclass constructor.

**Other Important points:**

* Call to super() must be first statement in Derived(Student) Class constructor.

For Example, The following will generate error:

**/\* superclass Person \*/**

**class Person**

**{**

**Person()**

**{**

**System.out.println("Person class Constructor");**

**}**

**}**

**/\* subclass Student extending the Person class \*/**

**class Student extends Person**

**{**

**Student()**

**{**

**System.out.println("Student class Constructor");**

**super();**

**}**

**}**

**/\* Driver program to test\*/**

**class SuperExample**

**{**

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

**{**

**Student s = new Student();**

**}**

**}**

* **If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error.** Object does have such a constructor, so if Object is the only superclass, there is no problem.
* If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called **constructor chaining.**
* You can use super only to access a method in a class’s superclass, not the superclass of the superclass. That is you cannot say **super.super.doStuff()**

**Initializer Block In Java:**In a Java program, operations can be performed on methods, constructors and initialization blocks. Instance Initialization Blocks or IIB are used to initialize instance variables. IIBs are executed before constructors. They run each time when object of the class is created.

Initialization blocks are executed whenever the class is initialized and before constructors are invoked.

They are typically placed above the constructors within braces.

It is not at all necessary to include them in your classes.

**1st Example:**

**// Java program to illustrate**

**// Instance Initialization Block**

**class GfG**

**{**

**// Instance Initialization Block**

**{**

**System.out.println("IIB block");**

**}**

**// Constructor of GfG class**

**GfG()**

**{**

**System.out.println("Constructor Called");**

**}**

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

**{**

**GfG a = new GfG();**

**}**

**}**

**Output:**

IIB block

Constructor Called

We can also have multiple IIBs in a single class. If compiler finds multiple IIBs, then they all are executed from top to bottom i.e. the IIB which is written at top will be executed first.

**2nd Example:**

**// Java program to illustrate**

**// execution of multiple**

**// Instance Initialization Blocks**

**// in one program**

**class GfG**

**{**

**// Instance Initialization Block - 1**

**{**

**System.out.println("IIB1 block");**

**}**

**// Instance Initialization Block - 2**

**{**

**System.out.println("IIB2 block");**

**}**

**// Constructor of class GfG**

**GfG()**

**{**

**System.out.println("Constructor Called");**

**}**

**// Instance Initialization Block - 3**

**{**

**System.out.println("IIB3 block");**

**}**

**// main function**

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

**{**

**GfG a = new GfG();**

**}**

**}**

**Output :**

IIB1 block

IIB2 block

IIB3 block

Constructor Called

**Rules For Constructor:**

* Constructors can use any access modifier (public, private, protected) including private. A private constructor means only code with in the class itself can instantiate an object of that type. So, if private constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows an instance created from within the class.   
    
  (private constructor or protected constructor both are used in singleton design pattern)
* The constructor name must match the name of the class
* Constructors must not have a return type.
* It’s legal (though stupid) to have a method with the same name as the class, but that does not make it a constructor. If you see a return type, it’s a method rather than constructor. In fact, you can have both a method and a constructor with the same name-the name of the class-in the same class. That’s not a problem in java (in c++, you are not allowed to that)
* If you don’t type a constructor to your class code, a default constructor will be automatically generated by the compiler.
* The default constructor is always a no arg constructor.
* If you want a no arg constructor and instead of overloading it, you have provided any other constructor (parametric or copy), compiler will not provide a no arg constructor any more.
* Every derived class constructor should have called to super class constructor using super keyword and it should be the first statement. if you explicitly use super keyword to call base class constructor, it should be the very first statement. Otherwise, it will give you compilation error.
* If you don’t call base class constructor from derived class constructor, compiler will include an implicit call to the base class’s no arg constructor. Now, if base class has no constructor, it is fine. (as compiler will include one by default) However, if base class provides other constructor but does not provide a default constructor, it will result a compilation error.
* A no arg constructor is not necessarily the default (that is, compiler supplied) constructor, although the default constructor is always a no arg constructor. Although the default constructor is always a no arg constructor, you are free to put in your own no arg constructor.
* Abstract class have constructors, and those constructors are always called when a concrete subclass is instantiated.
* Interfaces do not have constructors. Interfaces are not part of an object’s inheritance tree.
* The only way a constructor can be invoked is from within another constructor. In other words, you cant write code that actually calls a constructor as follows:  
  class Horse

{

Horse(){}

void doStuff()

{

Horse();

//calling the constructor-illegal

}

}

* You can always access a base class’s static member or static method from derived class’s non static method using super keyword, but, you can neither access base class’s non static variable or method nor access base class’s static method or static variable from Derived class’s static method.

**First Example:**

class Base

{

static int a;

static

{

a=10;

}

Base()

{

System.out.println("Base class constructor");

}

}

class Derived extends Base

{

Derived()

{

System.out.println("Derived class constructor");

}

void accessBaseStaticVariable()

{

System.out.println("The base class's a:"+super.a);

}

}

public class SuperExample2

{

public static void main(String argss[])

{

Derived derived=new Derived();

derived.accessBaseStaticVariable();

}

}

**This will compile fine and generate the following output:**

Base class constructor

Derived class constructor

The base class's a:10

**Second Example:**

class Base

{

int a;

Base()

{

a=10;

}

Base(int a)

{

this.a=a;

}

void seta(int a)

{

this.a=a;

}

}

class Derived extends Base

{

int b;

Derived()

{

b=10;

}

Derived(int b)

{

this.b=b;

}

static void setAOfBase(int a)

{

super.seta(a);

}

}

public class SuperExample3

{

public static void main(String args[])

{

Derived.setAOfBase(20);

}

}

**This will generate compilation error.**

SuperExample3.java:30: error: non-static variable super cannot be referenced from a static context

super.seta(a);

**Overloading and Overriding:**

**Overriding:**

Any time a class inherits a method form a super class, you have the opportunity to override the method**(unless, as you learned earlier, the method is final)**

The key benefit of overriding is to define the ability to define behaviour that’s specific to a particular subclass type.

For abstract methods you inherit from a superclass, you have no choice: but to implement the method in the subclass unless the subclass is also abstract. **(unlike c++, here in case of abstract class, abstract keyword is to be mentioned explicitly)**

**However, polymorphism, in which a base class reference holds the object of a subclass (it can be done based on the reference rule) is a tricky one.**

**In that case, the base class reference, cannot call the functions which are only specific to the subclass , whose object it is currently holding.**

Consider the following example:

**class Animal**

**{**

**public void eat()**

**{**

**System.out.println("Generic animal eating generically");**

**}**

**}**

**class Horse extends Animal**

**{**

**public void eat()**

**{**

**System.out.println("Horse eating hay, oats and horse treats");**

**}**

**public void buck()**

**{**

**}**

**}**

**public class PolymorphismTrial**

**{**

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

**{**

**Animal c=new Horse();**

**c.buck();**

**}**

**}**

This will generate a compilation error.

**Another important point** is overriding method cannot have a more restrictive access modifier than the method being overridden.

For instance, the following code will throw compilation error:

**class Animal**

**{**

**public void eat()**

**{**

**System.out.println("Generic animal eating generically");**

**}**

**}**

**class Horse extends Animal**

**{**

**private void eat()**

**{**

**System.out.println("Horse eating hay, oats and horse treats");**

**}**

**}**

**public class TestAnimals**

**{**

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

**{**

**Animal a=new Animal();**

**Animal b=new Horse();**

**a.eat();**

**b.eat();**

**}**

**}**

TestAnimals.java:10: error: eat() in Horse cannot override eat() in Animal

private void eat()

^

attempting to assign weaker access privileges; was public

1 error

**The basic overriding rules are the following:**

* The argument list must exactly match that of the overridden method, If they don’t match, you can end up with an overloaded method you did not intend.
* The return type must be same as, or a subtype of, the return type declared in in the original method in the superclass
* The access level of a overridden function cannot be more restricted than the original method.
* Instance methods can only be overridden only if they are inherited by the subclass.
* The overriding method can throw any unchecked exception, regardless of whether the original method declares the exception
* The overriding method must not throw any checked exception that are new or broader than those declared by the original method. For example, a method which declares a FileNotFoundException, cannot be overridden by a method which throws SQLException, Exception or any other runtime exception unless it is subclass of **FileNotFoundException**
* The overriding method can throw narrower and/or fewer exceptions.
* You cannot override a method marked as final.
* You cannot override a method that is a static. (because, static methods are class specific, not instance specific. Static methods can not be overridden. But can be overloaded since they are resolved using static binding by compiler at compile time.)

**Some More Conceptual Examples:**

**Example 1:**

class Animal

{

public void eat()

{

System.out.println("Animal's version of eat called");

}

}

class Dog extends Animal

{

public void eat()

{

System.out.println("Dog's version of eat called");

}

}

public class PolymorphismExample1

{

public static void main(String args[])

{

Dog labrador=new Dog();

labrador.eat();

}

}

Here Dog’s version of eat will be called. Though, in compile time, it will actually check if a function with no argument exists or not in Animal class. And, that function which is found in base class, it’s signature is kind of stored. So, in runtime, a function definition with the same function signature will be searched in derived class. If the definition is not found in derived class, it will invoke the function of base class.

**Example 2:**

class Animal

{

public void eat()throws Exception

{

throw new Exception("Animal's eat function was called");

}

}

class Dog extends Animal

{

public void eat()

{

System.out.println("Dog's eat function was called");

}

}

public class PolymorphismExample2

{

public static void main(String args[])

{

Animal a=new Dog();

Dog d=new Dog();

d.eat();

//it will generate no compilation error as d is a reference of Class Dog and is initialized with Dog object

a.eat();

//it will generate compilation error

}

}

If a method is overridden but you use a polymorphic (supertype) reference to refer to the subtype object with the overriding method, the compiler assumes that you are calling the supertype version of the method. If the supertype version declares a checked exception, but the overriding method does not, the compiler still thinks you are calling a method that declares an exception.

**Overloaded Methods:**

* overloaded method must change the argument list.
* Overloaded method can change the return type. (if you know either prototype concept or function signature concept, you will know, function signature depends upon the number of arguments, argument type,
* Overloaded method can change the access modifier.
* Overloaded methods can declare new or broader checked exception.
* A method can be overloaded in the same class or subclass. (however,the first rule is to be followed. Overloaded method must change the argument list)

Invoking overloaded methods that take object references rather than primitive type is a little more interesting. Say you have the overloaded method such that one version takes an animal and another version takes a horse (horse extends animal). If you pass a Horse object in the method invocation, you will invoke the overloaded version that takes Horse.

**class Animal**

**{**

**}**

**class Horse extends Animal**

**{**

**}**

**class UseAnimals**

**{**

**public void doStuff(Animal s)**

**{**

**System.out.println("In the animal version");**

**}**

**public void doStuff(Horse h)**

**{**

**System.out.println("In the horse version");**

**}**

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

**{**

**UseAnimals ua=new UseAnimals();**

**Animal animalObj=new Animal();**

**Horse horseObj=new Horse();**

**ua.doStuff(animalObj);**

**ua.doStuff(horseObj);**

**}**

**}**

Now, here, the output is what you expect

In the animal version

In the horse version

**But, what if you use an Animal reference to a horse object?**

**Animal animalRefToHorse=new Horse();**

**ua.doStuff(animalRefToHorse);**

Which of the overloaded versions is invoked? You might want to answer, “The want that takes a Horse, since, the reference to the Animal class **animalRefToHorse** is ultimately initialized with an object of Horse class.

Even though the actual object at runtime is a Horse and not an Animal. The choice of which overloaded method to call (in other words, the signature of the method) is not dynamically decided at runtime. It is decided at compile time. Hence, the upper example will print

In the animal version

**(in case of overloading)**

**Which overridden version of the method to call is decided at runtime based on the object type.**

**Animal animalRefToHorse=new Horse();**

Here, Animal is the reference type.

**new Horse(),** this Horse is the object type.

**class Animal**

**{**

**public void eat()**

**{**

**System.out.println("Generic animal eating generically"):**

**}**

**}**

**public class Horse extends Animal**

**{**

**public void eat()**

**{**

**System.out.println("Horse eating hay");**

**}**

**public void eat(String s)**

**{**

**System.out.println("Horse eating "+s);**

**}**

**}**

Notice that, the Horse class has both overloaded and overridden the eat() method.

**Combination Of Method Overloading and Overriding:**class Animal

{

public void eat()

{

System.out.println("Generic animal eating generically"):

}

}

public class Horse extends Animal

{

public void eat()

{

System.out.println("Horse eating hay");

}

public void eat(String s)

{

System.out.println("Horse eating "+s);

}

}

**Difference Between Overloaded Methods And Overriding Methods:**

|  |  |  |
| --- | --- | --- |
|  | **Overloaded Method** | **Overridden Method** |
| Argument(s) | Must change. | Must not change |
| Return Type | Can change | In most cases cannot change until the return type was an object to some superclass in the original method and object to the subclass in the overridden methods |
| Exceptions | Can change | Can reduce or eliminate. Can not throw new or broader checked exceptions |
| Access | Can change | Must not make more restrictive access |
| Invocation | Reference type determines which overloaded version is selected. Happens at compile time. | Object type (in other words, the type of the actual instance on the heap) determines which method is selected. Happens at runtime. |

**Polymorphism and Method Overloading and Method Overriding:**

|  |  |
| --- | --- |
| **Method Invocation Code** | **Result** |
| Animal a=new Animal();  a.eat(); | Generic animal eating generically |
| Horse h=new Horse(); h.eat(); | Horse eating hay |
| Animal ah=new Horse(); ah.eat(); | Horse eating Hay. Polymorphism works: The actual object type (Horse), not the reference type (Animal), is used to determine which eat() is called |
| Horse he=new Horse();  He.eat(“Apples”); | Horse eating Apples  The overloaded eat(String s) method is invoked |
| Animal a2=new Animal();  A2.eat(“treats”) | Compiler error!. Compiler does not see that the animal class does not have an eat() method that takes a string |
| Animal ah2=new Horse();  ah2.eat(“Carrots”); | Compiler error. Compiler will look only at the reference (compile time) and sees that Animal does not have an eat() method that takes a string. Compiler does not care the type of actual object. |

**Casting:**

You have seen how it’s both possible and common to use generic reference variable types to refer to more specific object types. (we could use a superclass reference to refer to subclass object types) It’s the heart of the polymorphism. For example, this line of code second nature by now:

**Animal animal=new Dog();**

But what happens when you want to use that animal reference variable to invoke a method which is specific to Dog. (and that method is not defined as a generic method is Animal)

In the following code, we have got an array of Animals, and whenever we find a Dog in the array, we want to do a special Dog thing.

class Animal

{

void makeNoise();

}

class Dog extends Animal

{

void makeNoise()

{

System.out.println("Bark");

}

void playDead()

{

System.out.println("Roll over");

}

}

public class CastTest

{

public static void main(String args[])

{

Animal []a={new Animal(), new Dog(), new Animal()};

for(Animal animal: a)

{

animal.makeNoise();

if(animal instanceof Dog)

//it will check the specific class Type of an object

{

animal.playDead();

}

}

}

}

It will necessarily generate compilation error.

However, if we change the following code block:

**if(animal instanceof Dog)**

**//it will check the specific class Type of an object**

**{**

**animal.playDead();**

**}**

To

**if(animal instanceof Dog)**

**//it will check the specific class Type of an object**

**{**

**Dog d=(dog) animal;**

**animal.playDead();**

**}**

It will work fine.

The new and improved code block contains a cast, which in this case is sometimes called a downcast, because, we are casting down the inheritance tree to a more specific class.

**Important Things About Casting:**

It’s important to know about that the compiler is forced to trust us when we do a downcast, even when we screw up:

**Class Animal**

**{**

**}**

**Class Dog extends Animal**

**{**

**}**

**Class DogTest**

**{**

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

**{**

**Animal animal=new Animal();**

**Dog d=(Dog)animal;**

**}**

**}**

The code will be compiled but it will fail later.

It will generate the ClassCastException at runtime.

java.lang.ClassCastException

Why cannot we trust the compiler to help use out here? Can’t it see that animal is of type Animal? All the compiler can do is to verify that the two types are in the same inheritance tree. So that depending upon whatever code might have come before the downcast, it’s possible that animal is of type Dog. The compiler must allow things that might possibly works at runtime. However, if compiler knows with certainty that the cast would not possibly work, compilation will fail. The following replacement code block will not compile:

**Animal animal=new Animal();**

**Dog d=(Dog) animal;**

**String a=(string) animal;**

In this case, you will get an error something like this: Inconvertible types.

Since, String and Animal do not belong to the same instance tree.

Now, unlike downcasting, upcasting (casting up the inheritance tree) works implicitly. Because, when you upcast you are implicitly restricting the number of methods you can invoke, as opposed to downcasting, which is implied that later on, you might want to invoke a more specific method.

The implicit typecast is always legal for assigning an object of a subtype to a reference of one of its supertype classes (or interfaces). If Dog implements pet **(even interface and class can be part of same inheritance tree? I understand that superclass and subclass are part of same inheritance tree)** and Pet defined beFriendly(), then a dog can be implicitly typecast to a pet, but the only Dog method you can invoke is beFriendly()

One more thing..If Dog implements Pet, then if Beagle extends Dog, but beagle does not declare that implements Pet, **it is completely Legal and Beagle is still a Pet. Beagle is a pet simply because it extends Dog, and Dog’s already taken care of the pet parts for itself, and for all its children. The Beagle class can always override any method it inherits from Dog, including methods that Dog implemented to fulfill its interface contract.**

**So, what do you learn?**

* When two objects belong to the same inheritance tree, the compiler will trust the developer though under some condition, it will generate run time error later.
* When two objects do not belong to the same inheritance tree, the casting is prevented by the compiler itself.

**Upcasting and Downcasting:**

Although there's no need to for programmer to upcast manually, it's allowed to do.

Consider the following example:

**Code:**

Mammal m = (Mammal)new Cat();

is equal to

**Code:**

Mammal m = new Cat();

But downcasting must always be done manually:

**Code:**

Cat c1 = new Cat();

Animal a = c1; //automatic upcasting to Animal

Cat c2 = (Cat) a; //manual downcasting back to a Cat

Why is that so, that upcasting is automatical, but downcasting must be manual? Well, you see, upcasting can never fail. But if you have a group of different Animals and want to downcast them all to a Cat, then there's a chance, that some of these Animals are actually Dogs, and process fails, by throwing ClassCastException.

This is where is should introduce an useful feature called "instanceof", which tests if an object is instance of some Class.

**Consider the following example:**

**Code:**

Cat c1 = new Cat();

Animal a = c1; //upcasting to Animal

if(a instanceof Cat){ // testing if the Animal is a Cat

System.out.println("It's a Cat! Now i can safely downcast it to a Cat, without a fear of failure.");

Cat c2 = (Cat)a;

}

Note, that casting can't always be done in both ways. If you are creating a Mammal, by calling "new Mammal()", you a creating a Object that is a Mammal, but it cannot be downcasted to Dog or Cat, because it's neither of them.

**For example:**

**Code:**

Mammal m = new Mammal();

Cat c = (Cat)m;

Such code passes compiling, but throws "java.lang.ClassCastException: Mammal cannot be cast to Cat" exception during running, because im trying to cast a Mammal, which is not a Cat, to a Cat.

**Java Memory Layout:**

**Java Heap Space**

**Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space.**

**Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.**

**Java Stack Memory**

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

**Difference between Java Heap Space and Stack Memory**

Based on the above explanations, we can easily conclude following differences between Heap and Stack memory.

Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.

Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.

Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.

Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at Java Garbage Collection.

Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.

We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory. We can use -Xss to define the stack memory size.

When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.

Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.

That’s all for Java Heap Space vs Stack Memory in terms of java application, I hope it will clear your doubts regarding memory allocation when any java program is executed.

**Some Deep Analyzing:**

**Consider the following example:**

class Collar

{

}

public class Dog

{

Collar c;

String name;

public static void main(String []args)

{

Dog d;

d=new Dog();

d.go(d);

}

void go(Dog dog)

{

c=new Collar();

dog.setName("aiko");

}

void setName(String dogName)

{

name=dogName;

//do more stuff

}

}

**Now, this is not the best example to learn java. But, it is a good example of learning stack and heap memory allocation.  
  
Line 7:** main() is placed on the stack

**Line 9:** Reference variable is created on the stack. But, there’s no Dog object yet.

**Line 10:** A new Dog object is created from heap and is assigned to the d reference variable (d reference variable is in the stack)

**Line 11:** A copy of the reference variable is passed to the go() method.

**Line 13:** the go method is placed on the stack, with the dog parameter as a local stack variable

**Line 14:** A new collar object is created to the heap and is assigned to Dog’s instance reference variable Collar c.

**Line 17:** setName() is addded to the stack, with the dogName parameter as it’s local variable

**Line 18:** The name instance variable now also refers to the String object.

**Now, note the following things:**

Notice that two different local variables refer to the same Dog object.

(d from main and dog from go)

Notice that one local variable and one instance variable both refer to the same String Aiko.

**After line 19th completes,** setName() completes and is removed from the stack. At this point, the local reference variable dogName disappears, too, although the String object is referred to is still on the top.

**Literals, Assignments And Variables:**

**Literal Values For All Primitive Types:**

'b': char literal

42: int literal

false: boolean literal

245789.343: double literal

**Integer Literal:**

**Numeric Literals with underscores:**you can define, but, don’t use underscore at the beginning or ending of the literal.

**Decimal Literals:  
  
Binary Literals:**

int b1=0B101010;//set b1 to binary 101010

int b2=0b0011;//set b2 to binary 11

**Octal Literals:**

**Hexadecimal Literals:**

**Floating Point Literals:**

Now, floating point literals are defined as a double by default. So, if you want to assign a floating point literal to a variable type of type float (32 bits), you must attach the suffix F or f.

**Boolean Literals:**

**Character literals:**

A char literal is represented by a single character in single question:

**char a= 'a';**

You can also assign a unicode value of the character. Using the Unicode notation of prefixing the value with '\u’ as follows:

**char letterN='\u004E';**

**Literal Values For Strings:**

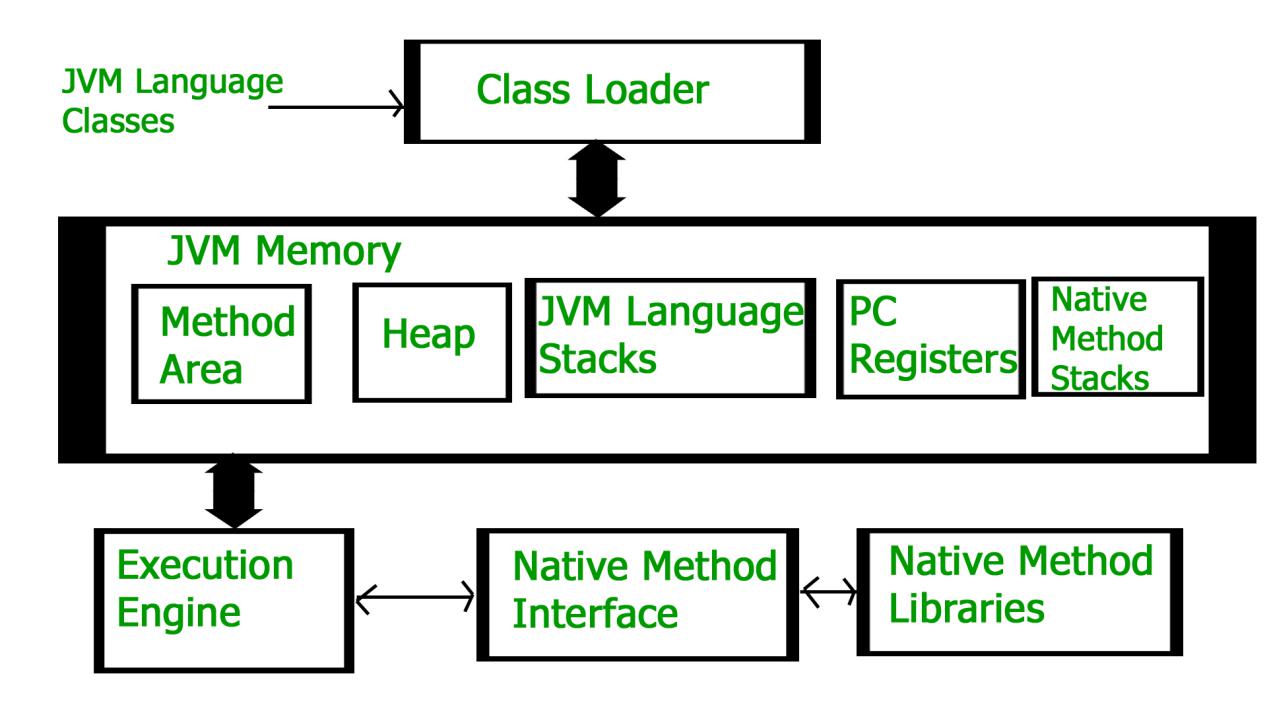
**Java Virtual Machine:**

**How JVM Works – JVM Architecture?**

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE(Java Run Environment).

Java applications are called WORA (Write Once Run Everywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a .java file, a .class file(contains byte-code) with the same filename is generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.



Now, as you can see (the lop layer is) class loader is on top.

**Class Loader SubSystem:**

It is mainly responsible for three activities.

Loading

Linking

Initialization

**Loading :**

The Class loader reads the .class file, generate the corresponding binary data and save it in method area. For each .class file, JVM stores following information in method area.

Fully qualified name of the loaded class and its immediate parent class.

Whether .class file is related to Class or Interface or Enum Modifier, Variables and Method information etc.

After loading .class file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in java.lang package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use getClass() method of Object class.

**getClass() method example and obtaining Information Example:**

// A Java program to demonstrate working of a Class type

// object created by JVM to represent .class file in

// memory.

import java.lang.reflect.Field;

import java.lang.reflect.Method;

// Java code to demonstrate use of Class object

// created by JVM

public class Test

{

public static void main(String[] args)

{

Student s1 = new Student();

// Getting hold of Class object created

// by JVM.

Class c1 = s1.getClass();

// Printing type of object using c1.

System.out.println(c1.getName());

// getting all methods in an array

Method m[] = c1.getDeclaredMethods();

for (Method method : m)

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

// getting all fields in an array

Field f[] = c1.getDeclaredFields();

for (Field field : f)

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

}

}

// A sample class whose information is fetched above using

// its Class object.

class Student

{

private String name;

private int roll\_No;

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public int getRoll\_no() { return roll\_No; }

public void setRoll\_no(int roll\_no) {

this.roll\_No = roll\_no;

}

}

**Linking:**

Performs verification, preparation, and (optionally) resolution.

**Verification :** It ensures the correctness of .class file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError.

**Preparation :** JVM allocates memory for class variables and initializing the memory to default values. (class variables: that means the static variables of a class)

**Resolution :** It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

**Initialization :**

In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed executed from top to bottom in a class and from parent to child in class hierarchy.

In general there are three class loaders :

**Bootstrap class loader :**

Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in JAVA\_HOME/jre/lib directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.

**Extension class loader :**

It is child of bootstrap class loader. It loads the classes present in the extensions directories JAVA\_HOME/jre/lib/ext(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.

**System/Application class loader :**

It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.

// Java code to demonstrate Class Loader subsystem

public class Test

{

public static void main(String[] args)

{

// String class is loaded by bootstrap loader, and

// bootstrap loader is not Java object, hence null

System.out.println(String.class.getClassLoader());

// Test class is loaded by Application loader

System.out.println(Test.class.getClassLoader());

}

}

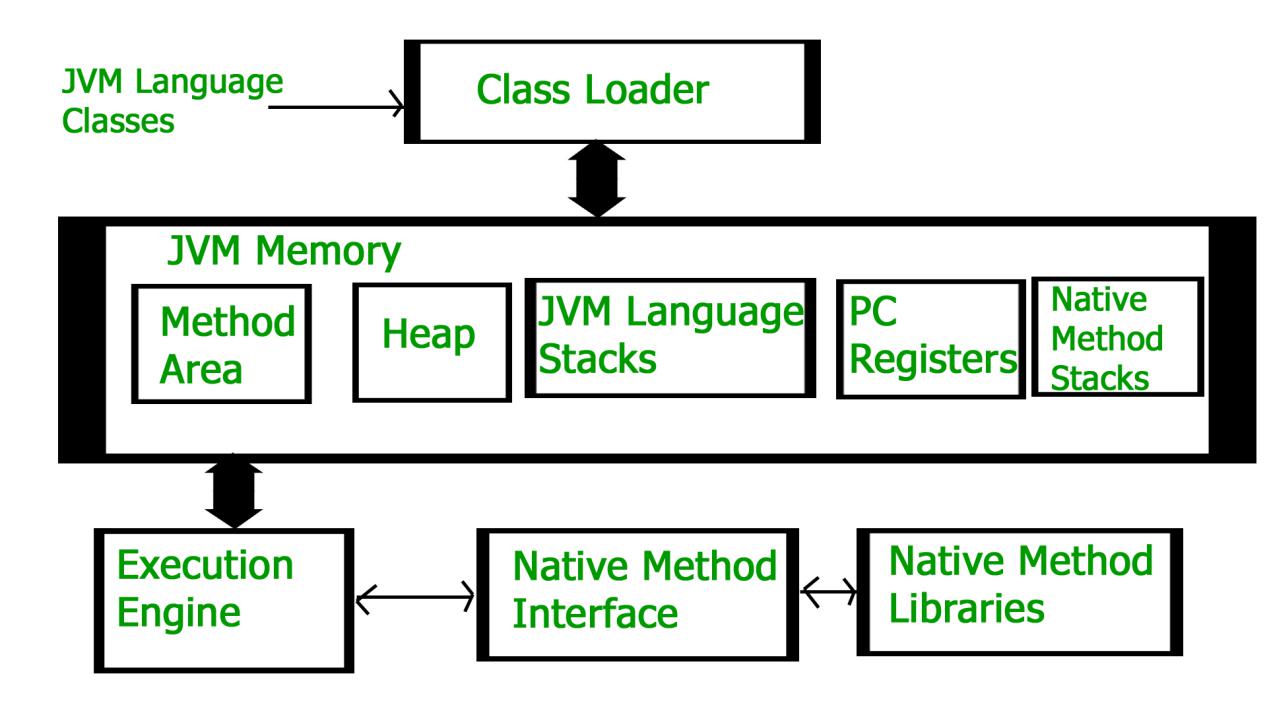
**Output:**

null

sun.misc.Launcher$AppClassLoader@73d16e93

**What are the classes loaded by ExtendedClassLoader?**

takes care of loading the extensions of the standard core Java classes. For instance, if you include log4j’s jar for logging, it will be added by the extended class loader.

****

**JVM Memory:**

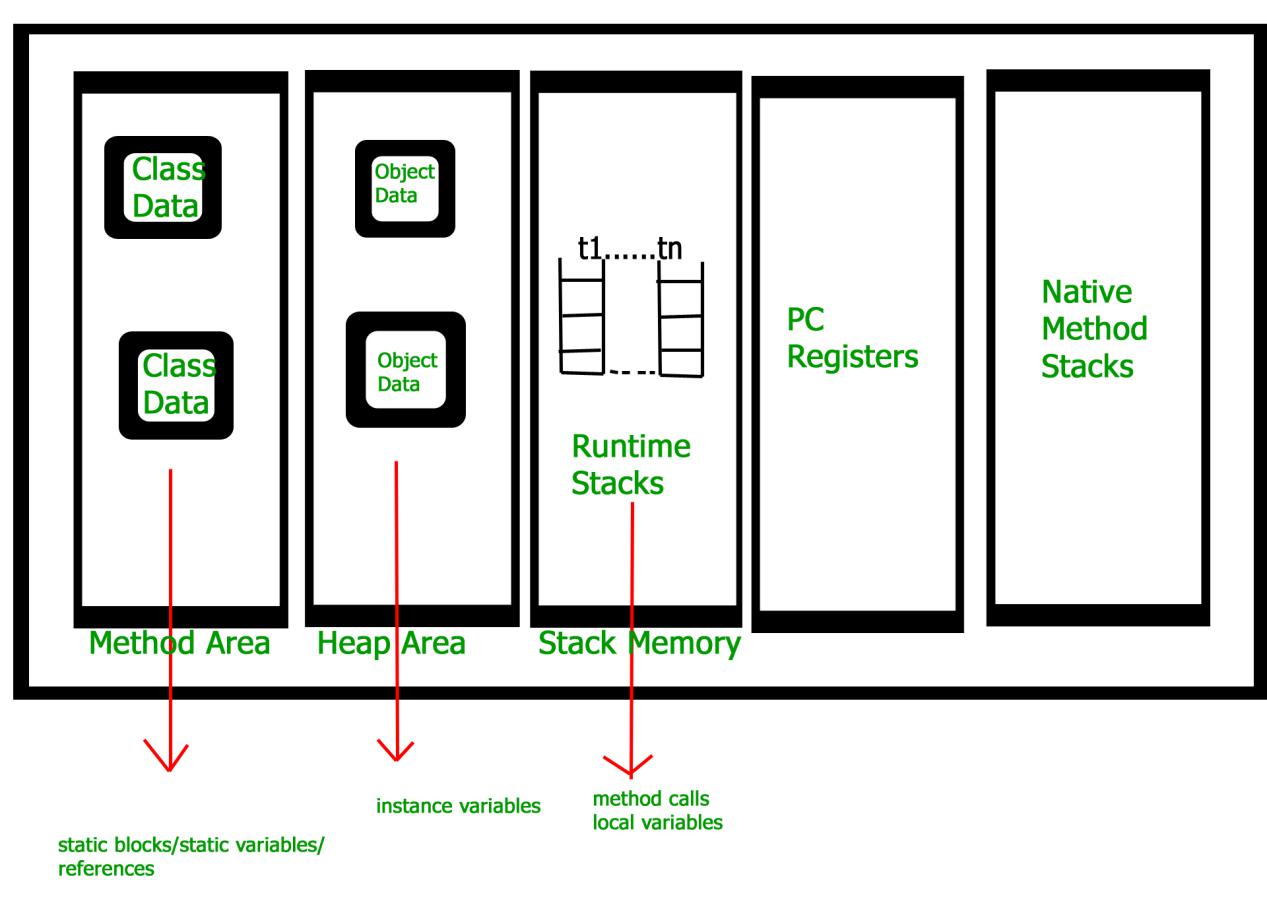
**Method area :**In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

**Heap area :**Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

**Stack area :**For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it’s run-time stack will be destroyed by JVM. It is not a shared resource.

**PC Registers :**Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

**Native method stacks :**For every thread, separate native stack is created. It stores native method information.



**Execution Engine:**

Execution engine execute the .class (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

**Interpreter :**

It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.

**Just-In-Time Compiler(JIT) :**

It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.

**Garbage Collector :**

It destroy un-referenced objects.

**Java Native Interface (JNI) :**

It is a interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries :**

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

**Garbage Collection in Java:**

n C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing OutOfMemoryErrors.

But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.

Garbage collector is best example of Daemon thread as it is always running in background.

Main objective of Garbage Collector is to free heap memory by destroying unreachable objects.

**Writing Code That Explicitly makes Objects Eligible For Collections:**

We can follow several techniques.

**Nulling a reference:**

**Reassigning a reference variable:**

**Isolating a reference:**

Now, this is a not so known term. (rest of the two can be easily understandable if you not new to programming)

This is another way in which objects can become eligible for garbage collection, even if they still have valid references. We call this scenario “Islands of isolation”.

A simple example is a class that has an instance variable that is a reference variable to another instance of the same class. Now imagine if two instances exist and they refer to each other. If all other references to these two objects are removed, even though each object still has a valid reference, there will be no way for any live thread to access either object.

**Forcing Garbage Collection:**

System.gc()  
  
Or,   
  
Runtime().getRuntime().gc()

**Some important points:**

These are requests to JVM. Not a command.

This method or these methods are suggested up to JAVA OCP 5. From java OCP 6, java garbage collector has evolved to such an advanced state that it is recommended you never invoke System.gc() in your code.

**Cleaning Up Before Garbage Collection: The finalize() method**

Java provides a mechanism that lets you run some code just before your object is deleted by the garbage collection. The code is located in a method named finalize() that all classes inherit from class object. On the surface, this sounds like a great idea, maybe your object opened up some resources, and you’d like to delete to close them before the object is deleted. But, the problem is that, you can never count on garbage collector() to delete an object. So, any code that you put into your class’s overridden finalize() method is not guaranteed to run.

**Some Points On finalize() method:**

**1.** Now, suppose you want to override finalize in java for a particular class. Now, ,making finalize protected looks good in terms of following rule of encapsulation which starts with least restrictive access modifier like private but making finalize private prevents it from being overridden in subclass as you can not override private methods, so making it protected is next obvious choice.

**2.** One of the most important points of finalize method is that it's not automatically chained like constructors. If you are overriding finalize method then it's your responsibility to call finalize() method of the superclass, if you forgot to call then finalize of super class will never be called. so it becomes critical to remember this and provide an opportunity to finalize of super class to perform cleanup. The best way to call superclass finalize method is to call them in the finally block as shown in below example. This will guarantee that finalize of the parent class will be called in all condition except when JVM exits:

@Override

protected void finalize() throws Throwable {

try{

System.out.println("Finalize of Sub Class");

//release resources, perform cleanup ;

}catch(Throwable t){

throw t;

}finally{

System.out.println("Calling finalize of Super Class");

super.finalize();

}

}

**3.** finalize method is called by garbage collection thread before collecting object and if not intended to be called like a normal method.

**4.** finalize gets called only once by GC thread if object revives itself from finalize method than finalize will not be called again.

**5.** Any Exception is thrown by finalize method is ignored by GC thread and it will not be propagated further, in fact, I doubt if you find any trace of it.

**6.**There is one way to increase the probability of running of finalize method by calling System.runFinalization() and Runtime.getRuntime().runFinalization(). These methods put more effort that JVM call finalize() method of all object which are eligible for garbage collection and whose finalize has not yet called. It's not guaranteed, but JVM tries its best.

**What not to do with finalize method in java:**

Trusting finalize method for releasing critical resource is biggest mistake java programmer can make. suppose instead of relying on close() method to release file descriptor, you rely on finalize to relapse it for you. Since there is no guaranteed when finalize method will run you could effectively lock hundreds of file-descriptor of earlier opened file or socket and there is high chance that your application will run out of file-descriptor and not able to open any new file. It's best to use finalize as the last attempt to do cleanup but never use finalize as a first or only attempt.

**Java Equality Operators For Primitive Types And For User Defined Types:**

**Now, we all know how equality operator (==, a relational operator) works for java primitive type.**

So, we will be discussing this for more complex examples:

**The equals Method in Class Object:**the equals() method in class object works in the same way that the == works (if not overridden)

If two reference variables point to the same object, equals will return true. If two references point to two different objects, even if they have the same values, the method will return false.

**The equals() method in String:**it is overridden for string class. So, if two references point to different objects, but those different objects have same values, equals will return true. Values are case sensitive here.

**Equality For enums (OCP only)**

Once, you have declared an enum, it’s not expandable. At runtime, there’s no way to make additional enum constants. Of course, you can have as many variables as you’d like, refer to a given enum constant, so, it’s important to be able to compare to two enums reference variables to see if they are equal- that is, do they refer the same enum constant. And both works fine.

**instanceof Comparison:**

The instanceof operator is used for reference variables only, and you can use it to check whether an object is of a particular type. By type, we mean class or interface type.

**First Example:**

**public class Example1**

**{**

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

**{**

**String s=new String("foo");**

**if(s instanceof String)**

**{**

**System.out.println("s is a string");**

**}**

**}**

**}**

**Prints this:**

s is a string.

**Now, a common example/common scenario where instanceof is used,**

Suppose, a base class reference is initialized with derived class object. In the runtime, we want to check base class’s object is initialized with which subtype’s object to do certain things. This is one of the most common scenario.

**Consider the following example:**

class A

{

}

class B extends A

{

public static void main(String []args)

{

A myA=new B();

m2(myA);

}

public static void m2(A a)

{

if(a instanceof B)

{

((B)a).dostuff();

//downcasting an A reference to a B Reference

}

}

public void dostuff()

{

System.out.println("a refers to B");

}

}

**The output will be the following:**

a refers to B

**Some more examples:**

**First Example:**

class A

{

}

class B extends A

{

public static void main(String []args)

{

B objB=new B();

if(objB instanceof A)

{

System.out.println("B is a derived class of A base class");

}

}

}

**This will print:**

B is a derived class of A base class

**Second Example:**

class A

{

}

class B extends A

{

public static void main(String []args)

{

A objA=new A();

if(objA instanceof B)

{

System.out.println("A is a derived class of B base class");

}

}

}

This will not print anything but will not throw any error either.

Because, A and B belong to the same inheritance tree.

Whether the following will generate compilation error.

**Example 3**

class A

{

}

class B

{

public static void main(String []args)

{

A objA=new A();

if(objA instanceof B)

{

System.out.println("A is a derived class of B base class");

}

}

}

**This will generate compilation error. Because, A and B do not belong to the same inheritance tree.**

**String and StringBuilder:**

**Strings are immutable object.**

What does that mean?

**String s=new String();**

We can do this. This line of code creates a new object of class string and assigns it to the variable a.

**s=”abcdef”**

Now, string class has a great number of constructors. You can use an efficient one:

**String s=new String(“abcdef”);**

And, there is even more concise:

**String a=”abcdef”**

**String s2=s;** //probably copy constructor

So far so good. String objects seem to be behaving just like other objects. So, what’s all the fuss about? Immutability. Once, you have assigned string a value that value will never change. It’s immutable. For instance, consider the following function:

**s=s.concat(“more stuff”);**

The concat method appends a literal to the end

Now, string objects are immutable. So, how is it allowed?   
  
The JVM took the value of s, (which was “abcdef”) and tacked “more stuff” onto the end, giving us the value. Now, since strings are immutable, the JVM could not stuff this new value into the old string referenced by s, so it created a new object, gave it the value “abcdef more stuff” and made a refer to it.

If you have not understood it, consider the following example:

**String s="Java":**

**x.concat(" Rules!");**

**System.out.println("x ="+x);  
  
Now, it’s output will be Java.** Let’s explain this: Create a new string object, give it the value “Java rules!” but nothing refers to it. The second string object is instantly lost (JVM plays it’s part), you cannot get to it. The reference variable x still refers to the original string with the value “Java”.

**String x="Java":**

**x.toUpperCase():**

**System.out.println("x="+x);**the output will still be “Java”

**Important Facts About Strings And Memory:**

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 Important Methods For Strings:  
  
charAt():** Returns the charcter located at the specified index

**concat():** Appends one string to the end of another.

**equals:**

**equalsIgnoreCase():** Determines the equality of two strings, ignoring case.

**length():**

**replace():**

**substring():**

**toLowerCase():**

**toString():** this is not a part of String class. But, important in case of converting other class’s object to String which are convertible (otherwise, you have to define your own toString method. That means overriding since, all classes implicitly extends Object class. And, in Object class there is a method toString(). And, since, it is a class method instead of being instance method, it will be overridden.

**The StringBuilder Class:**

The java.lang.StringBuilder class should be used when you have to make a lot of modifications to strings of characters. As discussed in the previous section, 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 string objects in the String pool. (JVM has a special place for storing string, right?)

A common use for **StringBuilder is File I/O when large, ever-changing streams of input are being handled by the program. In these cases, large blocks of characters are handled as units, and StringBuilder objects are the ideal way to handle a block of data, pass in on. And reuse the same memory to handle next block of data.**

**Now, let’s discuss the functions of StringBuilder:**

Now, it almost offers the same functionality as string in terms of functions. However, there are additional functions:

**void setCharAt(int index, char ch)**

The character at the specified index is set to ch.

(this is not present in String class. However, there are functions in String class, which are not present in the StringBuilder.

**Prefer StringBuilder to StringBuffer:**StringBuffer is thread safe (methods are thread safe). StringBuilder is not. Now, if we observe cases carefully and there are cases in which thread safety is not even needed, StringBuilder will run faster.

Now, all of the StringBuilder methods operate on the value of the StringBuilder object invoking the method. So, a call to sb.append(“def”) is actually appendng “def” to itself. In fact these methods can be chained to each other:

**StringBuilder sb=new StringBuilder(“abc”);**

**sb.append(“def”).reverse().insert(3,””---”);**

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

Now, a chain of methods which will act sequentially.

Append appends the def to abc. So, abcdef reverse reverses the string. fedcba

Insert(3,”---”) inserts “---” in the position 3.

**However, if you try to do it using String:**

String s=new String("abc");

s=s.concat("def").reverse().insert(3,"---");

//there is no append function

//reverse function is not there

//there is no insert function

Now, you will see there is a lot of functions which are not present there in String class. (which can potentially manipulates a string in place)

**Important Methods Of StringBuilder Class Which Are Not Present In String Class:**

**public StringBuilder append(String s)**

**public StringBuilder delete(int start,int end)**

**public StringBuilder insert(int offset,String s)**

**public StringBuilder reverse()**

**void setCharAt(int index,char ch)**

This functions are not present in String class.

**Arrays:**

* How to make an array reference variable
* How to make an array object(construct)
* How to populate the array with elements(initialize)

**Declaring An Array:**

**int[] key;**

**Int key[];**

Both are legal. However, second is more readable.

**Declaring an array of object references:**

**Thread[] threads;**

**Constructing An Array:**

**int []testScores;**//Declares the array of ints

**testScores=new int[4];**//constructs the array and assigns it to the testScores variable

**Initializing An Array:**

A reference that has not had an object assigned to it is a null reference.

**Initializing Elements In A Loop:**Array objects have a single public variable, length, that gives you the number of elements in the array. (In case of String, length() is a function. In case of an array, it is a variable. Because, in case of array, the length is variable.

**Dog[] myDogs=new Dog[6];**

**for(int x=0;x<myDogs.length;x++)**

**{**

**myDogs[x]=new Dog();**

**//assign a dog to index position**

**}**

**Constructing And Initializing An Anonymous Array:**

int[] testScores;

testScores=new int[]{4,7,2};

**Array Of Object References:**

If the declared array type is a class, you can put objects of any subclass of the declared type into the array. For example, if Subaru is a subclass of Car, you can put both Subaru objects and Car objects into an array of type car.

If the array is declared as an interface type, the array elements can refer to any instance of the class that implements the declared interface.

**ArrayList:**

**When To Use ArrayList:**

When you need to increase and decrease the size of you list of things.

The order of the things in your list is important and might change.

**Important methods in ArrayList class:**

**add(element):**

**add(index,element)**

**clear()**

**boolean contains(element)**

**Object get(index)**

**int indexOf(Object)**

**remove(index):**

**Handling Exceptions:**

**Catching Exception: Basic Try Catch Block:**

**Multiple Catch Block:**

**Finally Block (Actually finally clause):**

**Note:**

It is illegal to use a try clause without either a catch clause or a finally clause. A try clause by itself will result in a compiler issue. Any catch clause must immediately follow the try block. Any finally clause must immediately follow the last catch clause or it must immediately follow the try block if there is no catch. It is legal to omit either the catch clause or finally clause but not both.

**Throwing Multiple Exceptions:**

void myFunction() throws MyException1, MyException2

{

}

**Defining Your Own Exception:**The following is a very bad example. However, it’s still an example:

class MyException extends Exception

{

public MyException(String s)

{

// Call constructor of parent Exception

super(s);

}

}

// A Class that uses above MyException

public class Main

{

// Driver Program

public static void main(String args[])

{

try

{

// Throw an object of user defined exception

throw new MyException("GeeksGeeks");

}

catch (MyException ex)

{

System.out.println("Caught");

// Print the message from MyException object

System.out.println(ex.getMessage());

}

}

}

**Output**

Caught

GeeksGeeks

In the above code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception’s constructor using super(). The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

**(Because, compiler with provide a no arg constructor by defautt if you don’t not define any constructor. And, that constructor will call super() implicitly)**

**Example:**

// A Class that represents use-defined expception

class MyException extends Exception

{

}

// A Class that uses above MyException

public class setText

{

// Driver Program

public static void main(String args[])

{

try

{

// Throw an object of user defined exception

throw new MyException();

}

catch (MyException ex)

{

System.out.println("Caught");

System.out.println(ex.getMessage());

}

}

}

**Where Exceptions Come From:**

**JVM Exceptions:** those exceptions or errors are either exclusively or most logically thrown by JVM.

**Programmatic Exceptions:** these exceptions that are thrown explicitly by application and/or API programmers.

**JVM Thrown Exceptions:**

Like, NullPointerException

Consider the following example:

class NPE

{

static string s;

public static void main(String []args)

{

System.out.println(s.length());

}

}

Compiler will show no error. But, as soon the program runs, it will give NullPointerException error thrown my JVM.

**Programmatic Exceptions:**

**Rethrowing The Same Exception:**

Just you can throw an exception from a catch clause, you can also throw the same exception you just caught. But, remember in both cases the function should declare that it is throwing an exception in the function signature.

**Propagating An Uncaught Exception:  
  
(In c++, terminate function will call and the program will exist as default version of terminate calls abort)**

Now, you already know in java a try block is either followed by a catch block or finally block. You cannot omit both. So, why catch clauses are not mandatory?

Because, there is no requirement that you code a catch clause for every possible exception that could be thrown from the corresponding try block. If a method does not provide a catch clause for a particular exception, it is said to be ducking the exception. When the method finishes it’s execution and the is removed from the corresponding thread’s stack (from which is it called), it is expected to be caught by the function which is lying/residing in the stack top for the corresponding thread. If an exception reaches the bottom of the stack and main() even does not throw that exception, the program will stop. JVM will halt and the stack trace will be printed as output.

**Autoclosable Resources With A Try With Resources Statement:**

Now, we already learnt about finally block and we saw that finally block is a good place for closing files and assorted other resources. The examples made this clean up code in the finally block code look nice and short by writing.   
  
Unfortunately, real world clean up code is easy to get wrong. And when correct, it is verbose. Let’s check the following code to close our one resource when closing a file:

Reader reader=null;

try

{

//read from file

}

catch(IOException e)

{

log();

throw e;

}

finally

{

if(reader!=null)

{

reader.close();

}

catch(IOException e)

{

//ignore exception on closing the file

}

}

That’s a lot of code just to close a single file. But, that’s all necessary.

Now, to avoid this situation, developers typically write a helper class to close resources or they use the open resources.

Reader reader=null;

try

{

//read from file

}

catch(IOException e)

{

log();

throw e;

}

finally

{

HelperClass.close(reader);

}

Still, there are unnecessary lines.

Now, java 7 introduced a new feature called Automatic Resource management using “try-with-resources” to get rid of even these lines. The following code is equivalent to the previous example:

**try(Reader reader=new BufferedReader(new FileReader(file)))**

**{**

**//read from file**

**}**

**catch(IOException e)**

**{  
 log();**

**Throw e;**

**}**

No finally left at all. We wont even mention closing the reader. Automatic resource management takes care of it for us. **Let’s see what happen here. We start out by declaring the reader inside the try declaration.** **The parentheses are new. Think of them as a for loop in which we declare a loop index variable that is scoped to just the loop. Here, the reader is scoped to just the try block. Not the catch block. Just the try block.**

The actual try block does the same thing as before. It reads from the file. Or, at least, it comments that it would read from the file. The catch block also does the same thing. And just like our traditional try statement, catch is optional.

Now, you must be remembering the lesson where we mention that a try block must be either followed by catch statement or finally block or both. (i.e. we can not omit both of them). Time to learn something new about the rule.

We remember this is illegal code because it demonstrates a try without a catch or finally:

try

{

//do stuff

//need a catch or finally here

}

The following legal code demonstrates a try with resources with no catch or finally:

try(Reader reader=new BufferedReader(new FileReader(file)))

{

//do stuff

}

What’s the difference. The legal example does have a finally block, you just don’t see it. **The try with resources statement is logically calling a finally block to close the reader.** And, just to make things more trickier, you can add your own finally block to try with resources as well. Both will get called. We will take a look at how this works.

**Autocloseable and Closeable:**

closeable is an interface which was introduced in java 5. Whereas, Autocloseable is an interface introduced in java 7. (alongside with try with resources)   
  
Both provides the close block.

Closeable supports IOException and subclasses. Autocloseable supports any kind of exceptions.

What if we call close multiple times? For classes that implement Autocloseable, the implementation is required to be idempotent. Which means you can call close() all day and nothing will happen the second time and beyond. In case of closeable, there is no such guarantee.

**Assertion Mechanism:**

Assertions let you test your assumptions during development, without the expense (in both your time and program overhead) of writing exception handlers for exceptions that you assume will never happen once the program is out of development and fully deployed.

Assertions are typically enabled when an application is being tested and debugged, but disabled when the application is deployed. The assertions are still in code, although ignored by the JVM, so if you do have a deployed application that starts misbehaving, you can always choose to enable assertions in the field for additional testing.

**It is not always mandatory that assert expression must result in a boolean value. An assert expression could be used to print a variable’s value in the stack trace.**

**Enabling Assertions At Runtime:**

Run with -enableassertions or -ea option

**Disabling Assertions At Runtime:**

Run with -disableassertions or -da option.

We can mix them to make selective assertions in certain class:

java -ea -da:com.tourist

**Assertion Related General Standards:**

1. Don’t use assertions to validate arguments to a public method
2. Do use assertions to validate arguments to a private method
3. Don’t use arguments to validate command line arguments
4. Do use assertions, even in public methods, to check for cases, that you know are never ever supposed to happen.

**Autoclosable Resources With A try with resources statement:**

**Collections:**

There are a few basics operation which can be done using collection:

* Add objects to the collection
* Remove objects from the collection
* Find out if an object is in the collection
* Retrieve an object from the collection without removing it
* Iterate through the collection. Looking at each element (Object) one after another.

**Key Interfaces And Classes In The Collection FrameWork:**Maps

Sets

Lists

Queues

Utilities

**Now, differences:**

**collection(lowercase c):** which represents any of the data structures in which objects are stored and iterated over.

**Collection(capital C):** which is actually the java.util.Collection interface from which Set, List and Queue extend.

**Collections(capital C and ends with s):** is the java.util.Collections that holds a pile of static utility methods for use with collections.

**What is The Concept Of Ordered In Case Of ordered Collection:**

When a collection is ordered, it means you can iterate through the collection in a specific (not random) order.  
A hashtable collection is not ordered. Although the Hashtable itself has internal logic to determine the order. You wont find any order when you iterate through the hashtable. An ArrayList, however, keeps the order established by the element’s index position. Now, LinkedHashSet (as opposed to the ArrayList, where you can insert an element in a specific index position) keeps the order established by insertion.

**List Interface:**A list cares about the index. Key methods: get(int index), indexOf(Object 0), add(int index,Object o)

**Difference Between ArrayList And Vector:**

Vector is thread safe but ArrayList is not.

1. e. every method is synchronized. This is not so great though)

**LinkedList:**

A LinkedList is ordered by index position. The elements are doubly linked list to one another. The Linkage gives new methods for adding and removing from the beginning or end. The LinkedList class has been enhanced to implement the java.util.Queue interface. A such, it now supports the common queue methods peek(), poll() and offer().

**Set:**

A set cares about unqiueness. It does not allow duplicates. Now, check the equals function of Geek class to store it in set.

**class Geek**

**{**

**String name;**

**int id;**

**Geek(String name, int id)**

**{**

**this.name = name;**

**this.id = id;**

**}**

**@Override**

**public boolean equals(Object obj)**

**{**

**// if both the object references are**

**// referring to the same object.**

**if(this == obj)**

**return true;**

**// it checks if the argument is of the**

**// type Geek by comparing the classes**

**// of the passed argument and this object.**

**// if(!(obj instanceof Geek)) return false; ---> avoid.**

**if(obj == null || obj.getClass()!= this.getClass())**

**return false;**

**// type casting of the argument.**

**Geek geek = (Geek) obj;**

**// comparing the state of argument with**

**// the state of 'this' Object.**

**return (geek.name == this.name && geek.id == this.id);**

**}**

**@Override**

**public int hashCode()**

**{**

**// We are returning the Geek\_id**

**// as a hashcode value.**

**// we can also return some**

**// other calculated value or may**

**// be memory address of the**

**// Object on which it is invoked.**

**// it depends on how you implement**

**// hashCode() method.**

**return this.id;**

**}**

**}**

This is one of the proper ways to define Geek class so that it could be used with set.

There are three kinds of sets.   
  
**HashSet:**

**LinkedHashSet:**

**TreeSet:**

**Map Interface:**

A map cares about the unique identifier. You map a unique key (the id) to a specific value. Where both the key and value are, of course, objects.

**HashMap:**

**HashMap vs HashTable:**

Like, vector is synchronized counterpart of ArrayList, HashTable is a synchronized counterpart of HashMap. However, that means that the key methods of the class are synchronized.

**Autoboxing With Collections:**In general, java collections can hold objects but not primitives. Prior to java 5, a common use for so called “wrapper classes” (e.g:-Integer, Float, Boolean, and so on) to provide a way to get primitives into and out of collections.   
  
  
But, later versions of java, directly supports it:

**myInts.add(42);**

//autoboxing handles it

**Boxing, ==, Equals:  
  
Integer i3=10;**

**Integer i4=10;**

**If(i3==i4) System.out.println(“Same object”);**

**If(i3.equals(i4)) System.out.println(“Meantfully equal”);**

The example produces the output:

**Same object**

**Meantfully legal.**

Now, the equals() method should be working. But what happened with == and =?. Why is != telling us i1 and i2 are different objects. Whereas == iss saying that i3 and i4 are same objects. In order to save memory, two instances of the following wrapper objects will always be == when there primitive values are same.

**When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.**

Now, remember wrapper reference could be null. That means you have to watch out for code that appears to be safe primitive operations but that could throw a NullPointerException.

This code compiles fine, but the JVM throws a NullPointerException when it attempts to invoke dostuff(x). Because, x does not refer to an Integer object, sso there’s no value to unbox.

**Java.util.Collections Methods:**

* **Sorting Collections And Arrays:**

import java.util.\*;

public class SortingCollections

{

public static void main(String args[])

{

ArrayList<String> stuff=new ArrayList<String> ();

stuff.add("Denver");

stuff.add("Boulder");

stuff.add("Vail");

stuff.add("Aspen");

System.out.println("Unsorted: "+stuff);

Collections.sort(stuff);

System.out.println("Sorted: "+stuff);

}

}

**This is the basic sort method.**

* + **Sorting With comparable interface:**

The comparable interface is used by Collections.sort() method and the java.util.Arrays.sort() method to sort lists and arrays off objects respectively.

To implement Comparable, a class must implement a single method, compareTo()

Here’s an example of implementing Comparable interface:

**class DVDInfo implements Comparable<DVDInfo>**

**{**

**public int compareTo(DVDInfo d)**

**{**

**return title.compareTo(d.getTitle());**

**}**

**}**

**This can return three values:**

**Negative: (**if thisObject<anotherObject)

**Zero:**

**Positive:**

**When comparable interface is must required?** You are sorting a collection (array) of user defined object.

import java.util.\*;

class DVDInfo

{

String title;

String genre;

String leadActor;

DVDInfo(String t,String g,String a)

{

title=t;

genre=g;

leadActor=a;

}

public String toString()

{

return title+" "+genre+" "+leadActor+"\n";

}

}

class SortingExample1

{

public static void main(String args[])

{

ArrayList<DVDInfo> dvdList=new ArrayList<>();

dvdList.add(new DVDInfo("Donnie Darko","sci-fi","GyllenBall, Jake"));

dvdList.add(new DVDInfo("Rampage","Action/Adventure/Sci-fi","Johnson Dwayne"));

dvdList.add(new DVDInfo("Ready Player One","Action/Adventure/Sci-Fi","Pegg Simon"));

Collections.sort(dvdList);

for(DVDInfo currDVD: dvdList)

{

System.out.println(currDVD.toString());

}

}

}

Consider the example. If you compile it, you will get Compilation error.

**But the following will work properly.**

import java.util.\*;

class DVDInfo implements Comparable

{

String title;

String genre;

String leadActor;

DVDInfo(String t,String g,String a)

{

title=t;

genre=g;

leadActor=a;

}

public String getTitle()

{

return title;

}

public String toString()

{

return title+" "+genre+" "+leadActor+"\n";

}

public int compareTo(Object o)

{

DVDInfo d=(DVDInfo)o;

return title.compareTo(d.getTitle());

}

}

class SortingExample1

{

public static void main(String args[])

{

ArrayList<DVDInfo> dvdList=new ArrayList<>();

dvdList.add(new DVDInfo("Donnie Darko","sci-fi","GyllenBall, Jake"));

dvdList.add(new DVDInfo("Rampage","Action/Adventure/Sci-fi","Johnson Dwayne"));

dvdList.add(new DVDInfo("Ready Player One","Action/Adventure/Sci-Fi","Pegg Simon"));

Collections.sort(dvdList);

for(DVDInfo currDVD: dvdList)

{

System.out.println(currDVD.toString());

}

}

}

**This will compile fine and generate the following result.**

Donnie Darko sci-fi GyllenBall, Jake

Rampage Action/Adventure/Sci-fi Johnson Dwayne

Ready Player One Action/Adventure/Sci-Fi Pegg Simon

**But, you cannot do the following:**

import java.util.\*;

class DVDInfo implements Comparable

{

String title;

String genre;

String leadActor;

DVDInfo(String t,String g,String a)

{

title=t;

genre=g;

leadActor=a;

}

public String getTitle()

{

return title;

}

public String toString()

{

return title+" "+genre+" "+leadActor+"\n";

}

public int compareTo(DVDInfo d)

{

return title.compareTo(d.getTitle());

}

}

class SortingExample1

{

public static void main(String args[])

{

ArrayList<DVDInfo> dvdList=new ArrayList<>();

dvdList.add(new DVDInfo("Donnie Darko","sci-fi","GyllenBall, Jake"));

dvdList.add(new DVDInfo("Rampage","Action/Adventure/Sci-fi","Johnson Dwayne"));

dvdList.add(new DVDInfo("Ready Player One","Action/Adventure/Sci-Fi","Pegg Simon"));

Collections.sort(dvdList);

for(DVDInfo currDVD: dvdList)

{System.out.println(currDVD.toString());

}

}

}

* + **Sorting with comparator interface:**

Now, it is a little different. First, unlike Comparable interface, it Is not to be implemented by the Class whose objects we are sorting in an arraylist and try to sort. Rather, it would be a separate class implementing the Comparator interface and That class’s instance needs to be passed in Collections.sort method.

**The comparator interface is also very easy to implement,having only one method, compare(). (Which takes two arguments of the same class)**

**Difference Between Comparable And Comparator:**

1. Comparable provides single sorting sequence. In other words, we can sort the collection on the basis of single element such as id or name or price etc.

Comparator provides multiple sorting sequence. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.

2) Comparable affects the original class i.e. actual class is modified. Comparator doesn't affect the original class i.e. actual class is not modified.

3) Comparable provides compareTo() method to sort elements. Comparator provides compare() method to sort elements.

4) Comparable is found in java.lang package. Comparator is found in java.util package.

5) We can sort the list elements of Comparable type by Collections.sort(List) method. We can sort the list elements of Comparator type by Collections.sort(List,Comparator) method.

**Searching Array And Collections:**

**Arrays.binarySearch(sa,”one”);**

It is static method.

Ssa is the string array.

**Converting Array To List And Vice Versa:**

Arrays.asList() method.

myList.toArray() method.