**Source File Declaration Rules:**

●One public class per source code file

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

●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 we know more about it?**

public enum Constants {

ONE,

TWO,

THREE;

}

Compiling the above enum and disassembling resulting class file with javap gives the following: (Now, javap disassembles the machine code generated, as disassembling (Compiling the above enum and disassembling resulting class file with javap gives the following)

Compiled from "Constants.java"

**public final class Constants extends java.lang.Enum{**

**public static final Constants ONE;**

**public static final Constants TWO;**

**public static final Constants THREE;**

**public static Constants[] values();**

**public static Constants 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 they 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.**

**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