SUGGESTIONS

OOPS Basics: -

Definition: Object-oriented programming is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields, and code, in the form of procedures. A common feature of objects is that procedures are attached to them and can access and modify the object's data fields

Basic Properties or the principles of OOPs:

Inheritance- Inheritance is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation.

Polymorphism- Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

Abstraction- Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

Encapsulation- Encapsulation is an OOP concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Class and object:

A class is a non-primitive user defined data type in JAVA. A class description consists of a declaration and a definition. Usually, these pieces are split into separate files. An object is a single instance of a class. You can create many objects from the same class type.

Advantage of OOPs over conventional programming language:

1) Conventional programming divides the problem into functions but object oriented programming divides the problem into a number of entities called object.

2) Conventional programming represents non real modeling but object oriented programming represents real world modeling like the parents' child concept, inheritance.

3) Conventional programming follows a top-down approach but object oriented programming follow bottom-up approach.

4) Conventional programming does not support reusability but object oriented programming support reusability the code.

5) Conventional programming is used for designing medium sized application, maintenance and modification of large complex system is time consuming and costly. Object oriented programming is used for designing large and complex applications, maintenance and modification of large and complex system is relatively less time consuming and costly.

JAVA basics: -

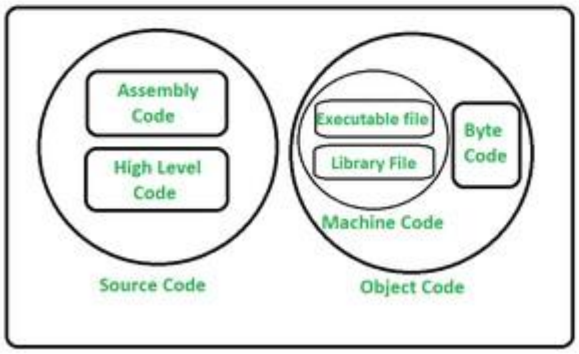
Source code: Source code refers to the high-level code or assembly code that is generated by a human/programmer. Source code is easy to read and modify. It is written by the programmer by using any High-Level Language or Intermediate language which is human-readable. Source code contains comments that the programmer puts for better understanding.

Source code is provided to the language translator which converts it into machine-understandable code which is called machine code or object code. The computer can not understand direct source code, the computer understands the machine code and executes it. It is considered as the fundamental component of the computer. In simple, we can say source code is a set of instructions/commands and statements which is written by a programmer by using a computer programming language like C, C++, Java, Python, Assembly language, etc. So the statements written in any programming language is termed as source code.

Byte code:

It is an intermediate code between the source code and machine code. It is a low-level code that is the result of the compilation of a source code which is written in a high-level language. It is processed by a virtual machine like Java Virtual Machine (JVM).
Byte code is a non-runnable code after it is translated by an interpreter into machine code then it is understandable by the machine. It is compiled to run on JVM, any system congaing JVM can run it irrespective of their Operating System. That's why Java is platform-independent. Byte code is referred to as a Portable code.

The below figure illustrates the arrangement of source code and byte code –



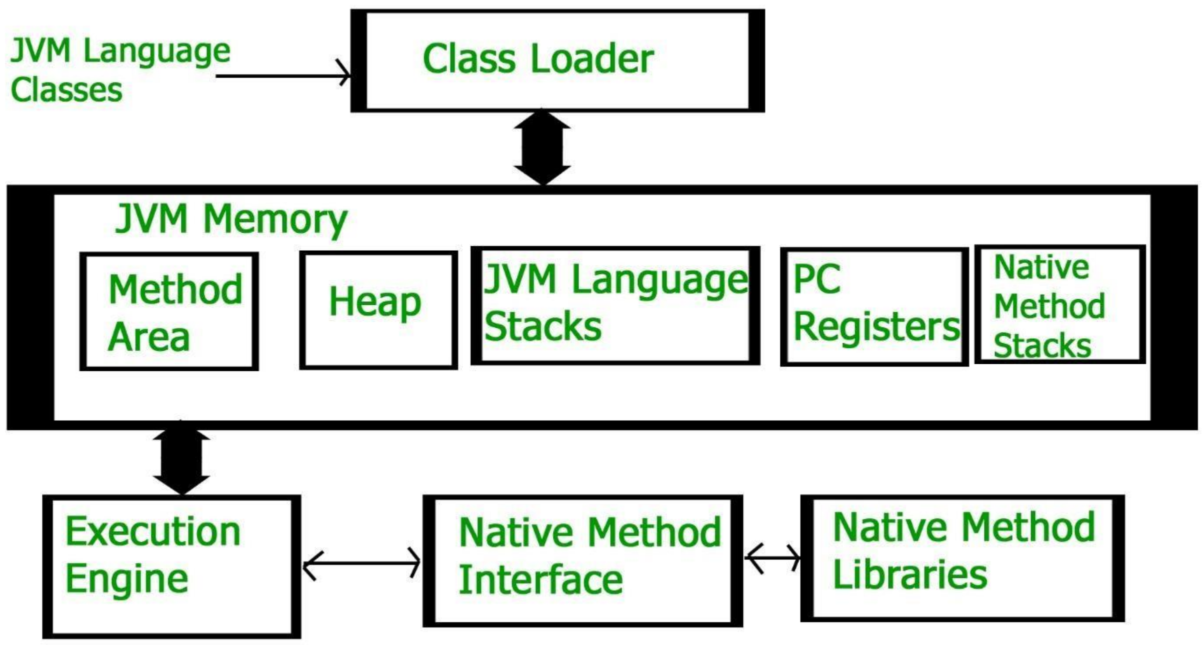
Difference between Source Code and Byte Code:

S.NO.	Source Code	Byte Code
01.	Source code is written by a human or programmer.	Byte code is not written by humans or programmers.

S.NO.	**Source Code**	**Byte Code**
02.	It is written by using some high-level programming language.	Byte code is an intermediate code between the source code and machine code.
03.	It is the input to the compiler and it is translated by the compiler or other language translator.	It is the input to the interpreter
04.	The source code is not directly understandable by the system/machine.	Byte code is executable by a virtual machine.
05.	The source code may contain comments.	Byte code does not contain comments.
06.	Source code is in the form of plain text similar to the English language.	Byte code is in the form of numeric codes and constants.
07.	Source code is more understandable by humans.	Byte code is less understandable by humans.
08.	Its speed is minimum than the byte code.	Its speed is maximum than the source code.
09.	The performance of source code is less than byte code.	The performance of the byte code is more than the source code.
10.	It is a high-level code.	It is an intermediate-level code.

JVM: A Java virtual machine is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation.

JVM architecture:



1) Class Loader
The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.
2) Method Area
JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.
3) Heap
All the Objects, their related instance variables, and arrays are stored in the heap. This memory is common and shared across multiple threads.
4) JVM language Stacks
Java language Stacks store local variables, and it's partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.
5) PC Registers
PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.
6) Native Method Stacks
Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.
7) Execution Engine
It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.
8) Native Method interface
The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.
9) Native Method Libraries
Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine.

How is Java platform independent?

Java is platform-independent because it uses a virtual machine. The Java programming language and all APIs are compiled into bytecodes. Bytecodes are effectively platform-independent. The virtual machine takes care of the differences between the bytecodes for the different platforms. The run-time requirements for Java are therefore very small. The Java virtual machine takes care of all hardware-related issues, so that no code has to be compiled for different hardware.

Encapsulation: -

Encapsulation is an OOP concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Data hiding Vs data abstraction:

Abstraction shows the essential information and hides the non-essential details. On the other hand, data hiding is used to hide the data from the components of the program by ensuring exclusive data access to class members.

The abstraction's purpose is to hide the complex implementation detail of the program or software. On the other hand, data hiding is implemented to achieve encapsulation.

Abstraction is used in class to create a new user-defined datatype. On the contrary, in classes data hiding is used to make the data private.

In abstraction abstract classes and interfaces are used whereas in data hiding getters and setters are used for the implementation.

Abstraction focuses on the observable behavior of the data whereas data hiding restricts or allow the use of data within a capsule.

Programs on this Topic

Inheritance :-

Definition: Inheritance is the mechanism of basing an object or class upon another object (prototypical inheritance) or class (class-based inheritance), retaining similar implementation.

Types:

Single inheritance: When a Derived Class to inherit properties and behavior from a single Base Class , it is called as single inheritance.

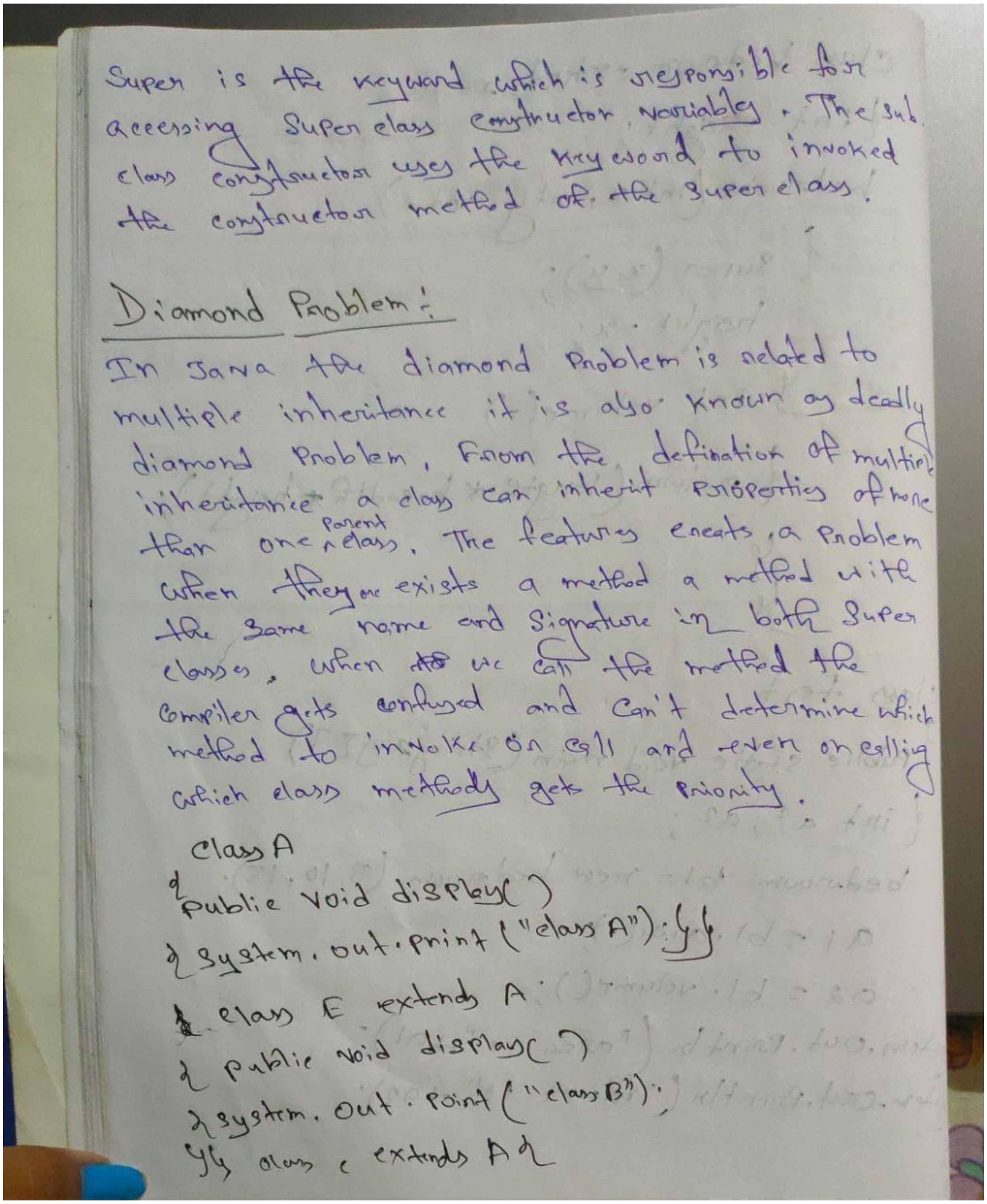
Multiple inheritance: Multiple inheritances allows programmers to create classes that combine aspects of multiple classes and their corresponding hierarchies.

Multilevel inheritance: A derived class is created from another derived class is called Multi Level Inheritance.

hybrid inheritance: Any combination of above three inheritance (single, hierarchical and multi level) is called as hybrid inheritance.

hierarchical inheritance: More than one derived classes are created from a single base class, is called Hierarchical Inheritance .

Diamond Problem:



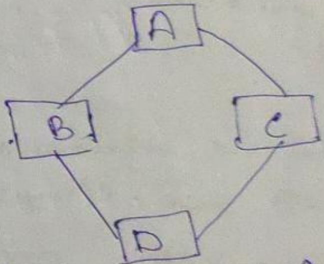
Super is the keyword which is responsible for accessing Super class constructor, variables. The sub class constructor uses the keyword to invoked the constructor method of the super class.

Diamond Problem:

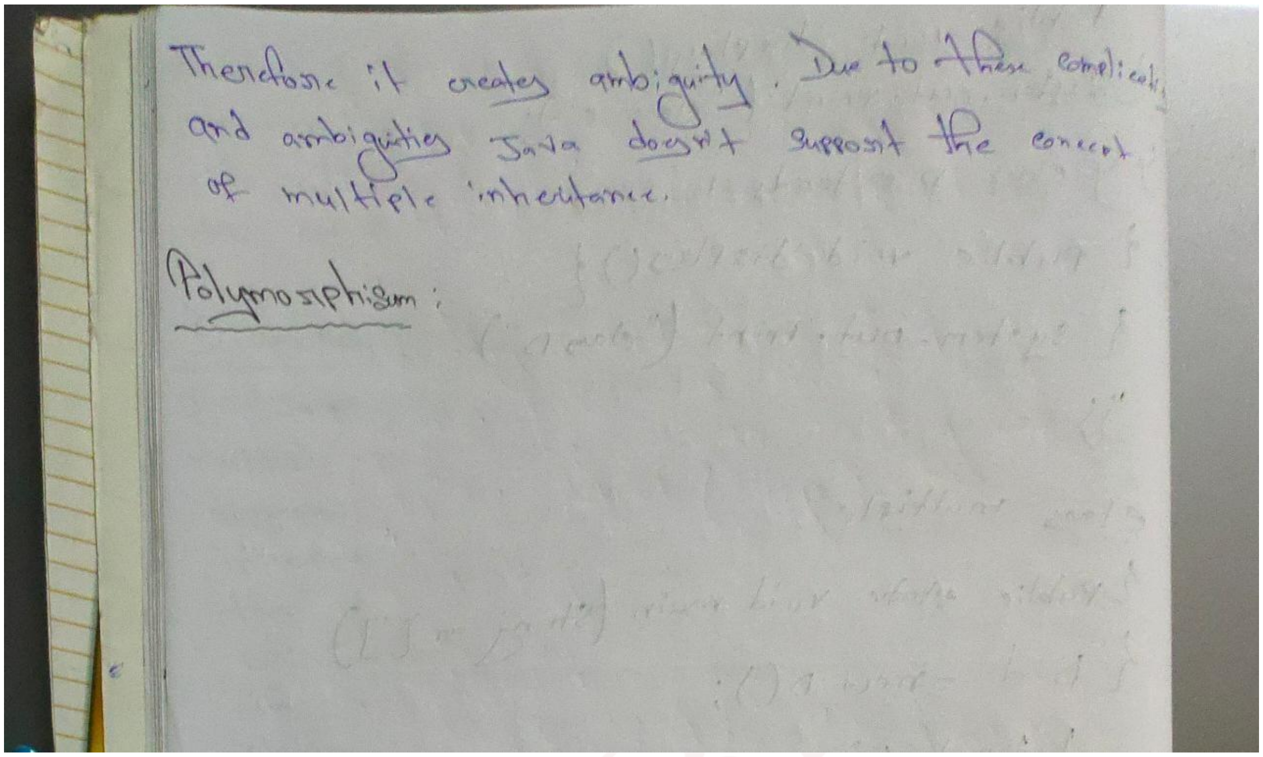
In Java the diamond Problem is related to multiple inheritance it is also known as deadly diamond Problem, From the defination of multiple inheritance a class can inherit properties of more than one parent class. The features creats a Problem when they are exists a method a method with the same name and Signature in both Super classes, when ~~to~~ we call the method the compiler gets confused and can't determine which method to invoke on call and even on calling which class methods gets the priority.

```
Class A
{
public void display( )
{ system.out.print ("class A"); } }
class B extends A
{ public void display( )
{ system.out.point ("class B");
} } class C extends A {
```

```
Public void display()
{System.out.print("class c");
}} class D extends B,
{ public void display(){
{ system.out.print ("class D")
}}

class multiple
{Public static void main (String ar[])
{D d =new D();
   d.display();
}
}.
```



class B & class c inherit class A. The display method of class A is ~~overridden~~ overwritten by class B and class c. Class ~~B~~D inherit class B and class c. Assum that we need to call the display method by using the object of class D. In these senario Java compiler does not know which display method to call



Therefore it creates ambiguity. Due to these complicati
and ambiguities Java doesn't support the concept
of multiple inheritance.

Polymorphism :

Final:

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared final, it can be assigned only once.

Program using Final keyword:

```
class Main {
   public static void main(String[] args) {

      // create a final variable final
      int AGE = 32;

      // try to change the final variable AGE =
      45;
      System.out.println("Age: " + AGE);
   }
}
```

Output

```
cannot assign a value to final variable AGE
      AGE = 45;
```

Polymorphism: -

Definition: Polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

Method Overloading: If a class has multiple methods having same name but different in parameters, it is known as Method Overloading**.**

Method Overriding: If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java**.**

Comparison between two methods:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide th* *specific implementation* of the method tha is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in tw* *classes* that have IS-A (inheritance relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Explain Dynamic Method Dispatch or Run time Polymorphism in JAVA

Dynamic method dispatch is the mechanism in which a call to an overridden method is resolved at run time instead of compile time. This is an important concept because of how Java implements run-time polymorphism.
Java uses the principle of ‘a superclass reference variable can refer to a subclass object’ to resolve calls to overridden methods at run time. When a superclass reference is used to call an overridden method, Java determines which version of the method to execute based on the type of the object being referred to at the time call.

In other words, it is the type of object being referred to that determines which version of an overridden method will be executed.

Advantages of dynamic method dispatch

1. It allows Java to support overriding of methods, which are important for run-time polymorphism.
2. It allows a class to define methods that will be shared by all its derived classes, while also allowing these sub-classes to define their specific implementation of a few or all of those methods.
3. It allows subclasses to incorporate their own methods and define their implementation.

```
// Implementing Dynamic Method Dispatch

class Apple
{
   void display()
   {
      System.out.println("Inside Apple's display method");
   }
}

class Banana extends Apple
{
   void display() // overriding display()
   {
      System.out.println("Inside Banana's display method");
   }
}

class Cherry extends Apple
{
   void display() // overriding display()
   {
      System.out.println("Inside Cherry's display method");
```

```
    }
}

class Fruits_Dispatch
{
    public static void main(String args[])
    {
        Apple a = new Apple(); // object of Apple
        Banana b = new Banana(); // object of Banana
        Cherry c = new Cherry();  // object of Cherry

        Apple ref;    // taking a reference of Apple

        ref = a; // r refers to a object in Apple
        ref.display(); // calling Apple's version of display()

        ref = b;   // r refers to a object in Banana
        ref.display(); // calling Banana's version of display()

        ref = c;  // r refers to a object in Cherry
        ref.display(); // calling Cherry's version of display()
    }
}
```

Output:

```
Inside Apple's display

method Inside Banana's

display method Inside

Cherry's display method
```

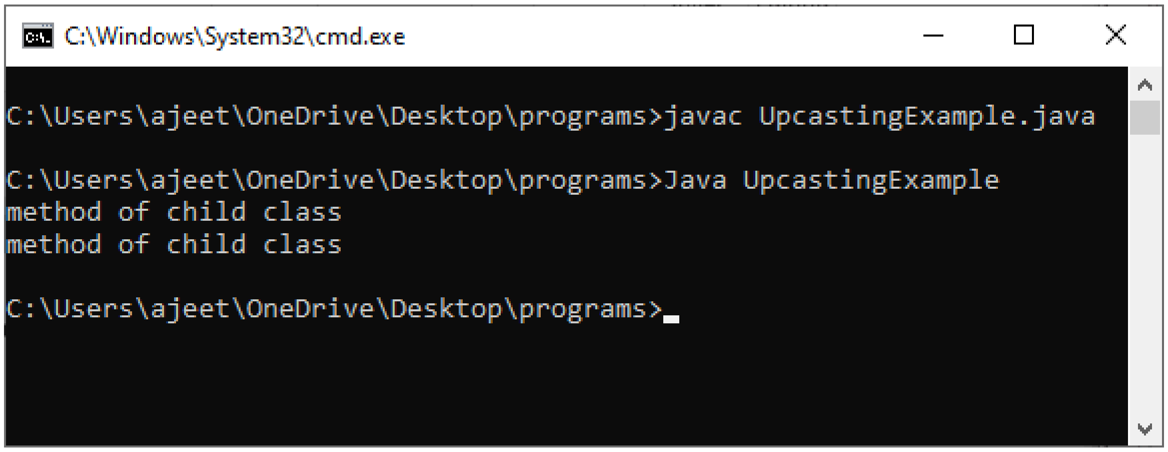
Upcasting

Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class. Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. Upcasting is also known as Generalization and Widening.

UpcastingExample.java

```
class Parent{
   void PrintData() {
      System.out.println("method of parent class");
   }
}

class Child extends Parent {
  void PrintData() {
    System.out.println("method of child class");
  }
}
class UpcastingExample{
  public static void main(String args[]) {

    Parent obj1 = (Parent) new Child();
    Parent obj2 = (Parent) new Child();
    obj1.PrintData();
    obj2.PrintData();
  }
}
```

Output:



C:\Windows\System32\cmd.exe

```
C:\Users\ajeet\OneDrive\Desktop\programs>javac UpcastingExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>Java UpcastingExample
method of child class
method of child class

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

Downcasting:

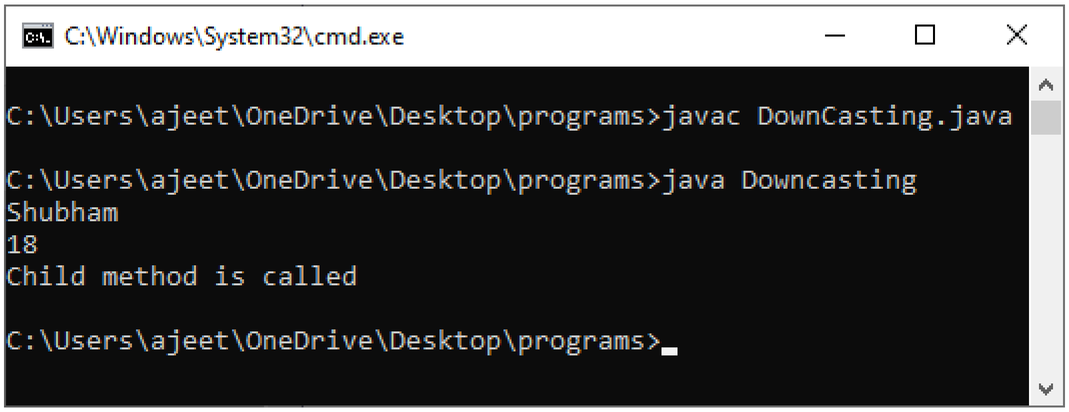
Upcasting is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the "ClassCastException". Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

Below is an example of downcasting in which both the valid and the invalid scenarios are explained:

DowncastingExample.java

```
//Parent class
class Parent {
    String name;

    // A method which prints the data of the parent class
    void showMessage()
    {
        System.out.println("Parent method is called");
    }
}

// Child class
class Child extends Parent {
    int age;

    // Performing overriding
    @Override
    void showMessage()
    {
        System.out.println("Child method is called");
    }
}

public class Downcasting{

    public static void main(String[] args)
    {
        Parent p = new Child();
        p.name = "Shubham";

        // Performing Downcasting Implicitly
        //Child c = new Parent(); // it gives compile-time error

        // Performing Downcasting Explicitly
        Child c = (Child)p;

        c.age = 18;
        System.out.println(c.name);
        System.out.println(c.age);
        c.showMessage();
    }
}
```

Output:



```
C:\Users\ajeet\OneDrive\Desktop\programs>javac DownCasting.java

C:\Users\ajeet\OneDrive\Desktop\programs>java Downcasting
Shubham
18
Child method is called

C:\Users\ajeet\OneDrive\Desktop\programs>
```

Miscellaneous: -

Abstract Class: A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Interface: An interface in Java is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction.
There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in JAVA.
In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
Java Interface also represents the IS-A relationship.
It cannot be instantiated just like the abstract class.
Since Java 8, we can have default and static methods in an interface.
Since Java 9, we can have private methods in an interface.

Difference between Abstract class and Interface:

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.
9)**Example:** public abstract class Shape{	**Example:** public interface Drawable{

public abstract void draw(); }	void draw(); }

Solution of multiple inheritance:/Abstraction using interface/WAP to prove that JAVA supports multiple inheritance with the help of interface.

```
interface AnimalEat {
  void eat();
}
interface AnimalTravel {
  void travel();
}
class Animal implements AnimalEat, AnimalTravel {
  public void eat() {
    System.out.println("Animal is eating");
  }
  public void travel() {
    System.out.println("Animal is travelling");
  }
}
public class Demo {
  public static void main(String args[]) {
    Animal a = new Animal();
    a.eat();
    a.travel();
  }
}
```

Output

```
Animal is eating
Animal is travelling
```

Wrapper Class: The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Use of Wrapper classes in Java:

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- Synchronization: Java synchronization works with objects in Multithreading.
- java.util package: The java.util package provides the utility classes to deal with objects.
- Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

String and String buffer class:

No.	String	StringBuffer
1)	The String class is immutable.	The StringBuffer class is mutable.
2)	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate t strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4)	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.
5)	String class uses String constant pool.	StringBuffer uses Heap memory

Garbage Collection: Garbage collection in Java is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

How Does Garbage Collection in Java works?

Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Finalize method: The Java finalize () method of object class is a method that the garbage collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection, or we can say resource de-allocation. Remember, it is not a reserved keyword. Once the finalize () method completes immediately, Garbage Collector destroys that object.

Finalization: Just before destroying any object, the garbage collector always calls finalize () method to perform clean-up activities on that object. This process is known as Finalization in Java.

Note: The Garbage collector calls the finalize () method only once on any object.

Syntax:

protected void finalize throws Throwable {}

Since the Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class, Garbage Collector can call the finalize () method on any java object.

Why finalize() method is used?
finalize() method releases system resources before the garbage collector runs for a specific object. JVM allows finalize() to be invoked only once per object.

How to override finalize() method?
The finalize method, which is present in the Object class, has an empty implementation. In our class, clean-up activities are there. Then we have to override this method to define our clean-up activities.
In order to Override this method, we have to define and call finalize within our code explicitly.

- Java

```
// Java code to show the
// overriding of finalize() method

import java.lang.*;

// Defining a class demo since every java class
// is a subclass of predefined Object class
// Therefore demo is a subclass of Object class
public class demo {

    protected void finalize() throws Throwable
    {
        try {

            System.out.println("inside demo's finalize()");
        }
        catch (Throwable e) {

            throw e;
        }
        finally {

            System.out.println("Calling finalize method"
                             + " of the Object class");

            // Calling finalize() of Object class
            super.finalize();
        }
    }

    // Driver code
    public static void main(String[] args) throws Throwable
    {

        // Creating demo's object
        demo d = new demo();

        // Calling finalize of demo
        d.finalize();
    }

}
```

Output:
inside demo's finalize()

Calling finalize method of the Object class

static members: In Java, a *static* member is a member of a class that isn't associated with an instance of a class. Instead, the member belongs to the class itself. As a result, you can access the static member without first creating a class instance.

```
public class StaticMemberProgram{

    static int StaticTotalLength(String a, String b) {
        // Add up lengths of two strings.
        return a.length() + b.length();
    }

    static int StaticAverageLength(String a, String b) {
        // Divide total length by 2.
        return StaticTotalLength(a, b) / 2;
    }

    public static void main(String[] args) {

        // Call methods.
        int total = StaticTotalLength("Golden", "Bowl");
        int average = StaticAverageLength("Golden", "Bowl");

        System.out.println(total);
        System.out.println(average);
    }
}
```

Now as You can see in the above example to call methods StaticTotallength & StaticAverageLength , we didn't create any class object.

The output of the above program will be

```
10

5
```

There are two types of static members.

Static field: A static field is in programming languages is the declaration for a variable that will be held in common by all instances of a class. The static modifier determines the class variable as one that will be applied universally to all instances of a particular class. A final modifier can also be added to indicate that the class variable will not change.

A static field may also be called a class variable.

Static method: A static method (or *static function*) is a method defined as a member of an object but is accessible directly from an API object's constructor, rather than from an object instance created via the constructor.

What are nested methods?

If a method is defined inside another method, the inner method is said to be nested inside the outer method or it is called Nested method. All languages do not support nesting a method inside another method but python allows you to do so.
JAVA does not support "directly" nested methods. But it can still be achieved by 3 methods:
Method 1 (Using anonymous subclasses), Method 2 (Using local classes), Method 3 (Using a lambda expression)

Thread life cycle: defines its potential states. During any given point of time, the thread can only be in one of these states:

NEW – a newly created thread that has not yet started the execution

RUNNABLE – either running or ready for execution but it's waiting for resource allocation

BLOCKED – waiting to acquire a monitor lock to enter or re-enter a synchronized block/method

WAITING – waiting for some other thread to perform a particular action without any time limit

TIMED_WAITING – waiting for some other thread to perform a specific action for a specified period

TERMINATED – has completed its execution.

Package: -

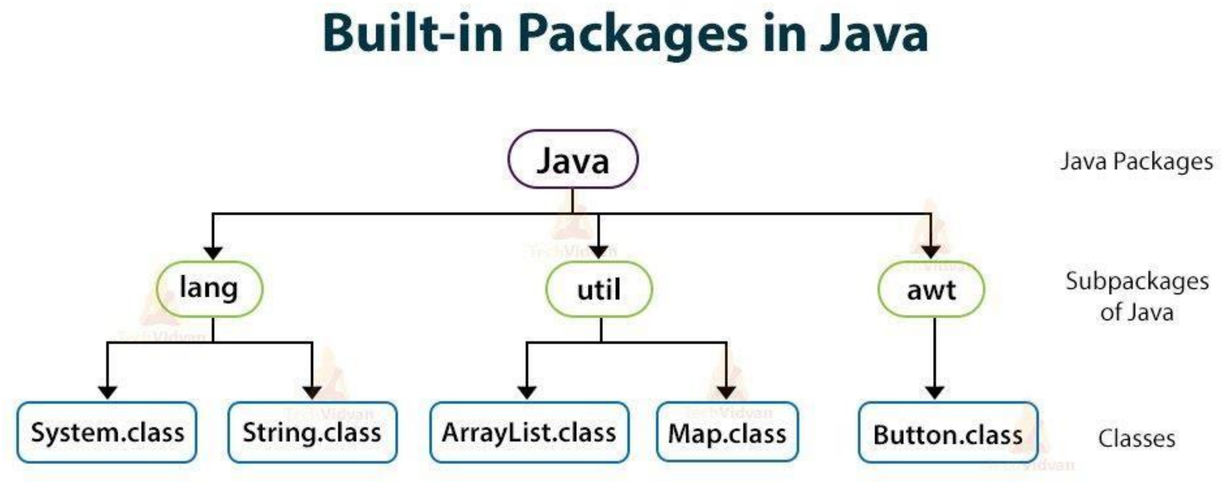
Definition: A Java package organizes Java classes into namespaces, providing a unique namespace for each type it contains. Classes in the same package can access each other's package-private and protected members.

How many types of packages are there?

1. Java API packages or built-in packages

Java provides a large number of classes grouped into different packages based on a particular functionality.

Examples:
java.lang: It contains classes for primitive types, strings, math functions, threads, and exceptions.
java.util: It contains classes such as vectors, hash tables, dates, Calendars, etc.
java.io: It has stream classes for Input/Output.
java.awt: Classes for implementing Graphical User Interface – windows, buttons, menus, etc.
java.net: Classes for networking
java. Applet: Classes for creating and implementing applets



2. User-defined packages

As the name suggests, these packages are defined by the user. We create a directory whose name should be the same as the name of the package. Then we create a class inside the directory.

Import keyword used for what?

import keyword is used to import built-in and user-defined packages into our Java program.

Package keyword used for what?

The package keyword is used to specify a directory structure to which the current source file must belong.

How many access specifiers are there in JAVA? Mention the limitations of different access specifiers:

Java provides four types of access specifiers. Private, Default, protected, public.

#1) Default: Whenever a specific access level is not specified, then it is assumed to be 'default'. The scope of the default level is within the package.
#2) Public: This is the most common access level and whenever the public access specifier is used with an entity, that particular entity is accessible throughout from within or outside the class, within or outside the package, etc.
#3) Protected: The protected access level has a scope that is within the package. A protected entity is also accessible outside the package through inherited class or child class.
#4) Private: When an entity is private, then this entity cannot be accessed outside the class. A private entity can only be accessible from within the class.

Access Specifier	**Inside Class**	**Inside Package**	**Outside package subclass**	**Outside package**
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

Exception Handling: -

Definition: In computing and computer programming, exception handling is the process of responding to the occurrence of exceptions – anomalous or exceptional conditions requiring special processing – during the execution of a program.

Five keywords: Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.

Keyword	**Description**
Try	We specify the block of code that might give rise to the exception in a special block with a "Try" keyword.
Catch	When the exception is raised it needs to be caught by the program. This is done using a "catch" keyword. So a catch block follows the try block that raises an exception. The keyword catch should always be used with a try.

Keyword	Description
Finally	Sometimes we have an important code in our program that needs to be executed irrespective of whether or not the exception is thrown. This code is placed in a special block starting with the "Finally" keyword. The Finally block follows the Try-catch block.
Throw	The keyword "throw" is used to throw the exception explicitly.
Throws	The keyword "Throws" does not throw an exception but is used to declare exceptions. This keyword is used to indicate that an exception might occur in the program or method.

Difference between throw and throws:

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.

5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Multi catch: A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Nested try: As the name suggests, a try block within a try block is called nested try block in Java. This is needed when different blocks like outer and inner may cause different errors. To handle them, we need nested try blocks.

User defined exception: Java user-defined exception is a custom exception created and throws that exception using a keyword 'throw'. It is done by extending a class 'Exception'. An exception is a problem that arises during the execution of the program.

Programs on these topics:

Applet: -

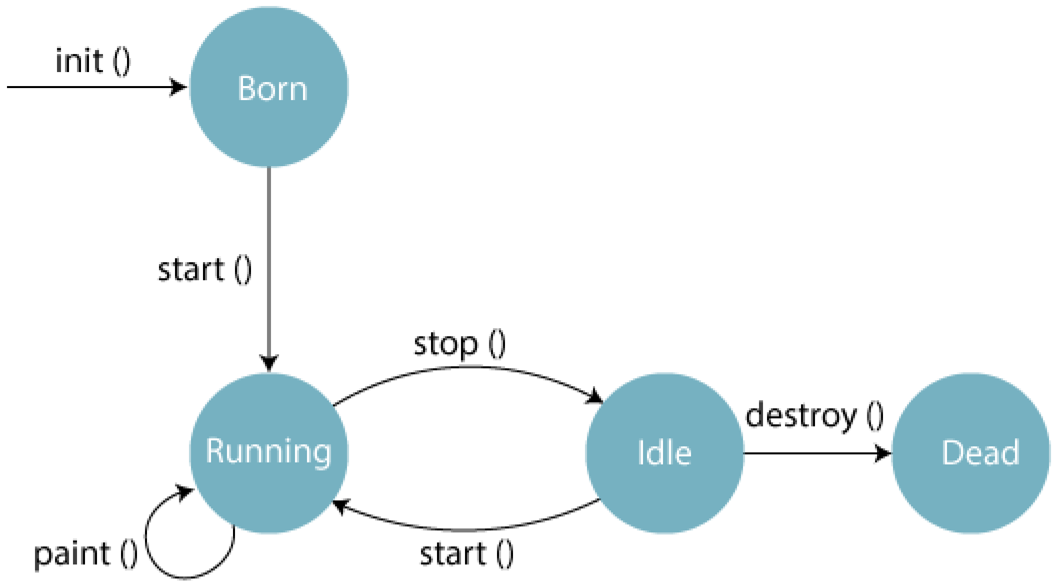
Definition: An applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server. Applets are used to make the website more dynamic and entertaining.

Difference with main program:

Java Application	Java Applet
Applications are just like a Java programs that can be execute independently	Applets are small Java programs that are designed to be included with the HTML web document. They

Java Application	Java Applet
without using the web browser.	require a Java-enabled web browser for execution.
Application program requires a main function for its execution.	Applet does not require a main function for its execution.
Java application programs have the full access to the local file system and network.	Applets don't have local disk and network access.
Applications can access all kinds of resources available on the system.	Applets can only access the browser specific services. They don't have access to the local system.
Applications can executes the programs from the local system.	Applets cannot execute programs from the local machine.
An application program is needed to perform some task directly for the user.	An applet program is needed to perform small tasks or the part of it.

life cycle of applet:



Applet Life Cycle Working:

The Java plug-in software is responsible for managing the life cycle of an applet.

An applet is a Java application executed in any web browser and works on the client-side. It doesn't have the main() method because it runs in the browser. It is thus created to be placed on an HTML page.

The init(), start(), stop() and destroy() methods belongs to the applet.Applet class.

The paint() method belongs to the awt.Component class.

In Java, if we want to make a class an Applet class, we need to extend the Applet

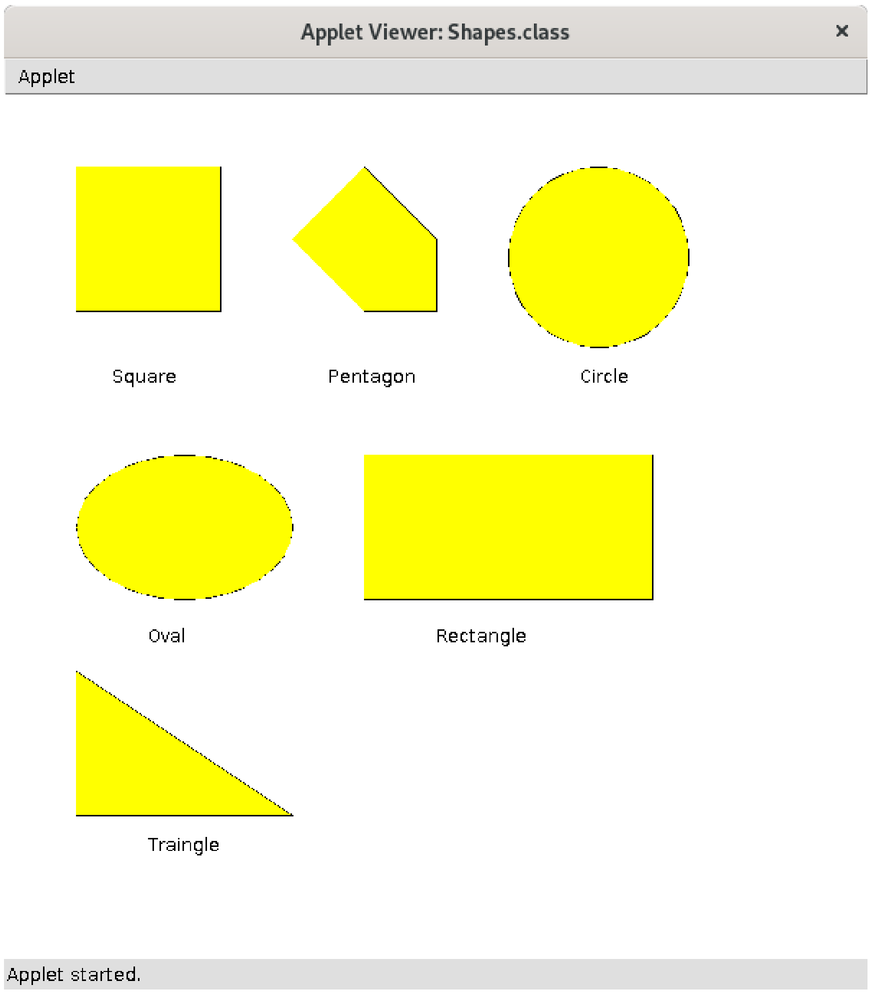
Whenever we create an applet, we are creating the instance of the existing Applet class. And thus, we can use all the methods of that class.

programs of drawing different colored geometric figures:

```
/*Java Program to Create and Fill Shapes using Applet*/
import java.applet.*;
import java.awt.*;
public class Shapes extends Applet
{
    //Function to initialize the applet
    public void init()
    {
 setBackground(Color.white);
            }
            //Function to draw and fill shapes
```

```java
    public void paint(Graphics g)
    {
        //Draw a square
        g.setColor(Color.black);
        g.drawString("Square",75,200);
        int x[]={50,150,150,50};
        int y[]={50,50,150,150};
        g.drawPolygon(x,y,4);
        g.setColor(Color.yellow);
        g.fillPolygon(x,y,4);
        //Draw a pentagon
        g.setColor(Color.black);
        g.drawString("Pentagon",225,200);
        x=new int[]{200,250,300,300,250,200};
        y=new int[]{100,50,100,150,150,100};
        g.drawPolygon(x,y,6);
        g.setColor(Color.yellow);
        g.fillPolygon(x,y,6);
        //Draw a circle
        g.setColor(Color.black);
        g.drawString("Circle",400,200);
        g.drawOval(350,50,125,125);
        g.setColor(Color.yellow);
        g.fillOval(350,50,125,125);
        //Draw an oval
        g.setColor(Color.black);
        g.drawString("Oval",100,380);
        g.drawOval(50,250,150,100);
        g.setColor(Color.yellow);
        g.fillOval(50,250,150,100);
        //Draw a rectangle
        g.setColor(Color.black);
        g.drawString("Rectangle",300,380);
        x=new int[]{250,450,450,250};
        y=new int[]{250,250,350,350};
        g.drawPolygon(x,y,4);
        g.setColor(Color.yellow);
        g.fillPolygon(x,y,4);
        //Draw a triangle
        g.setColor(Color.black);
        g.drawString("Traingle",100,525);
        x=new int[]{50,50,200};
        y=new int[]{500,400,500};
        g.drawPolygon(x,y,3);
        g.setColor(Color.yellow);
        g.fillPolygon(x,y,3);
    }
}
/*
<applet code = Shapes.class width=600 height=600>
</applet>
*/
```

Output:



What are the conditions for creating constructor in JAVA?

A **constructor** is used to initialize an object when it is created. It is syntactically similar to a method. The difference is that the constructors have same name as their class and, have no return type.

There is no need to invoke constructors explicitly these are automatically invoked at the time of instantiation.

Rules to be remembered

While defining the constructors you should keep the following points in mind.

- A constructor does not have return type.
- The name of the constructor is same as the name of the class.
- A constructor cannot be abstract, final, static and Synchronized.
- You can use the access specifiers public, protected & private with constructors