Dr.SNS RAJALAKHMI COLLEGE OF ARTS AND SCIENCE

(AUTONOUMOUS)

DEPARTMENT OF COMUTER APPLICATIONS

Course Name & Code : **OBJECT ORIENTED PROGRAMMING -21UCU403**

Staff Name : **Ms.R.Kavipriya**

Class : **I BCA B**

Semester : **II**

Year : **2022-2023 Even**

UNIT-III

Classes and Objects in JAVA

OBJECT	CLASS
Object is an instance ofa class.	Class is a blue print from which objects are created
Object is a real world entity such as chair, pen. table, laptop etc.	Class is a group of similar objects.
Object is a physical entity.	Class is a logical entity.
Object is created many times as per requirement.	Class is declared once.
Object allocates memory when it is created.	Class doesn't allocated memory when it is created.
Object is created through new keyword. Employee ob = new Employee();	Class is declared using class keyword. class Employee{}
There are different ways to create object in java:- New keyword, newinstance() method, clone() method, And deserialization.	There is only one way to define a class, i.e., by using class keyword.

Example: Java Class and Objects

```
class Lamp
{
   boolean isOn;
  void turnOn() {
     isOn = true;
    System.out.println("Light on? " + isOn);
  }
  void turnOff() {
    isOn = false;
    System.out.println("Light on? " + isOn);
  }
}
class Main {
  public static void main(String[] args) {
    Lamp led = new Lamp();
```

```
    Lamp halogen = new Lamp();
    led.turnOn();



    halogen.turnOff();
  }
}
```

Run Code

Output:

```
Light on? true
Light on? false
```

In the above program, we have created a class named `Lamp`. It contains a variable: `isOn` and two methods: `turnOn()` and `turnOff()`.

Inside the `Main` class, we have created two objects: `led` and `halogen` of the `Lamp` class. We then used the objects to call the methods of the class.

- **led.turnOn()** - It sets the `isOn` variable to `true` and prints the output.
- **halogen.turnOff()** - It sets the `isOn` variable to `false` and prints the output.

The variable `isOn` defined inside the class is also called an instance variable. It is because when we create an object of the class, it is called an instance of the class. And, each instance will have its own copy of the variable.

That is, `led` and `halogen` objects will have their own copy of the `isOn` variable.

2. What is Strings in JAVA

Java String

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

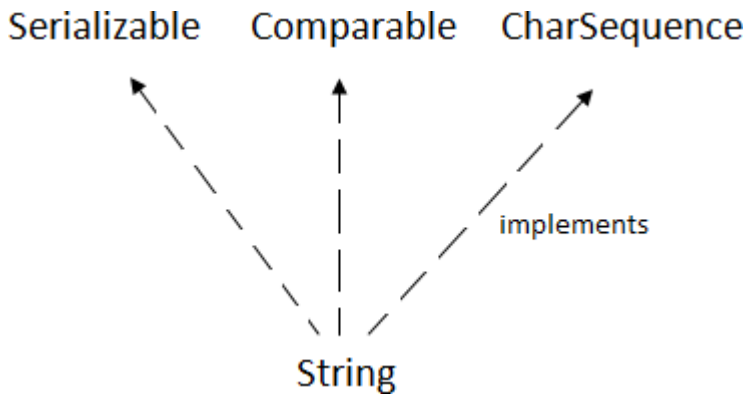
```
1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);
```

is same as:

```
1. String s="javatpoint";
```

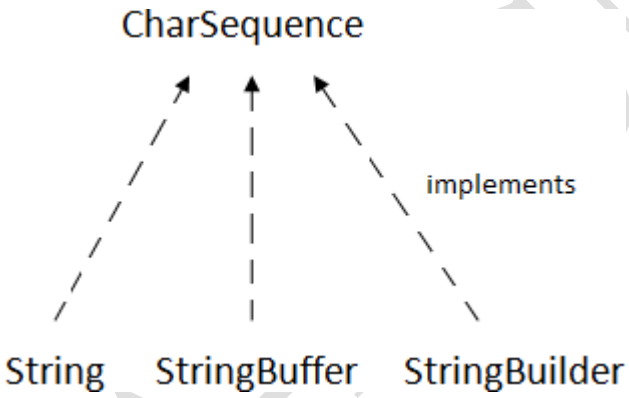
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what String in Java is and how to create the String object.

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

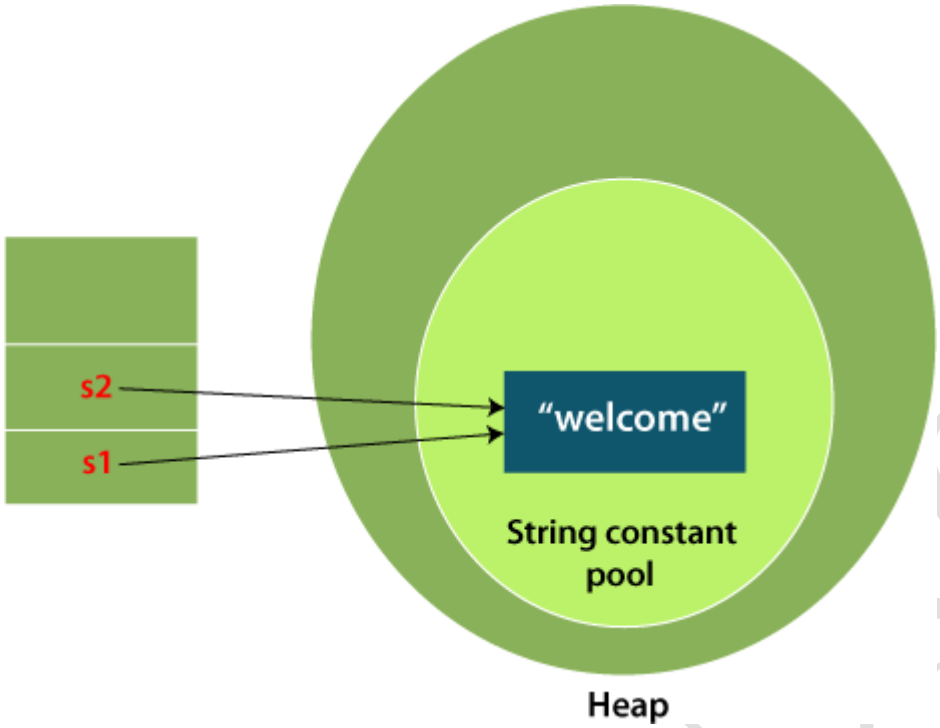
1) String Literal

Java String literal is created by using double quotes. For Example:

```
1. String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance
```



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Java String Example

StringExample.java

```
1. public class StringExample{
2. public static void main(String args[]){
3. String s1="java";//creating string by Java string literal
4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string
6. String s3=new String("example");//creating Java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10.}}
```

Output:

```
java
strings
example
```

The above code, converts a *char* array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

3. What is Inheritance and its type

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs

(Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes

that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

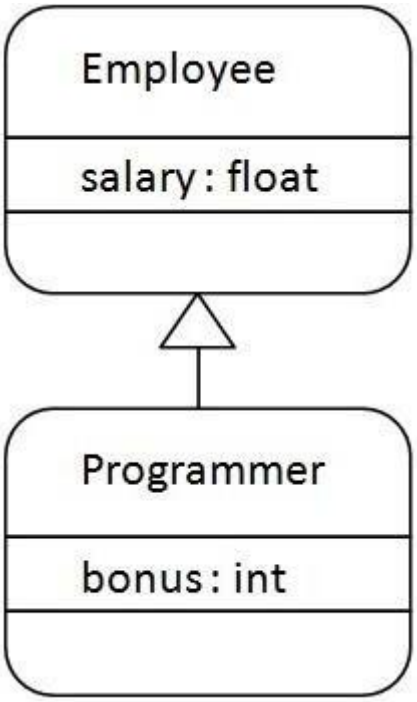
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
1. class Subclass-name extends Superclass-name
2. {
3.     //methods and fields
4. }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2.  float salary=40000;
3. }
```

```
4.  class Programmer extends Employee{
5.   int bonus=10000;
6.   public static void main(String args[]){
7.     Programmer p=new Programmer();
8.     System.out.println("Programmer salary is:"+p.salary);
9.     System.out.println("Bonus of Programmer is:"+p.bonus);
10.}
11.}
```

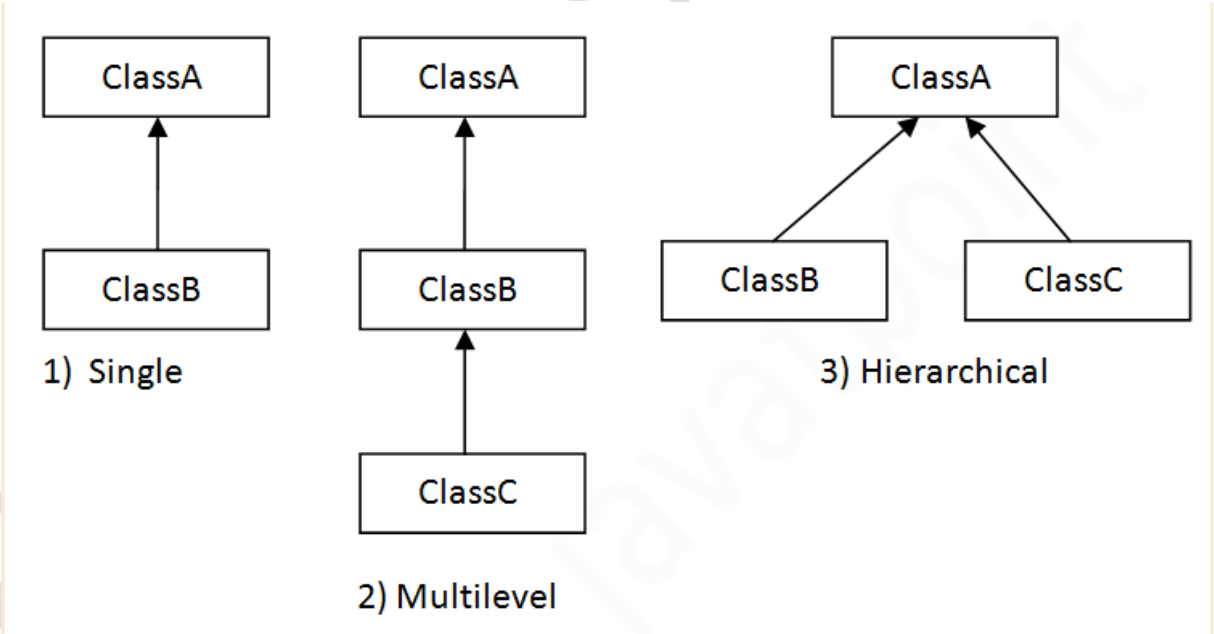
```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

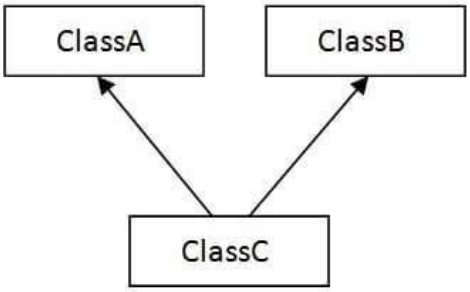
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, **multiple and hybrid inheritance is supported through interface only**. We will learn about interfaces later.

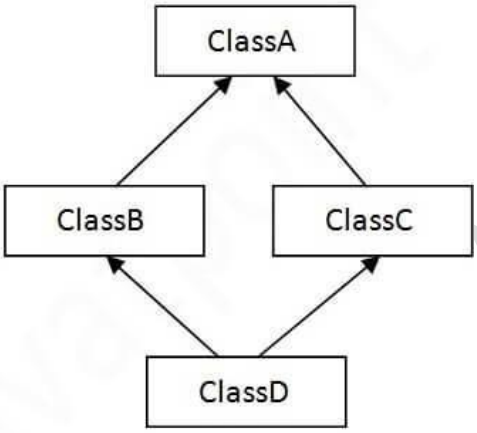


Note: Multiple inheritance is not supported in Java through class.

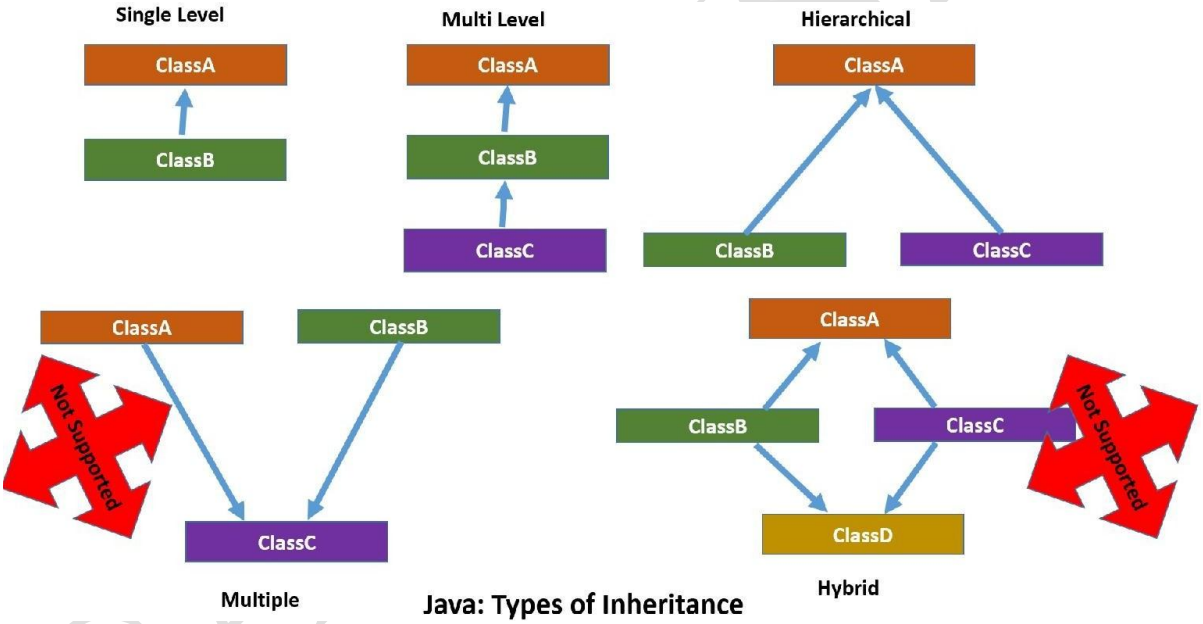
When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



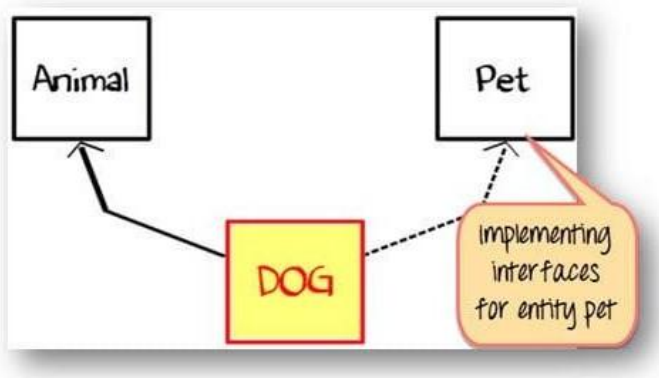
5) Hybrid

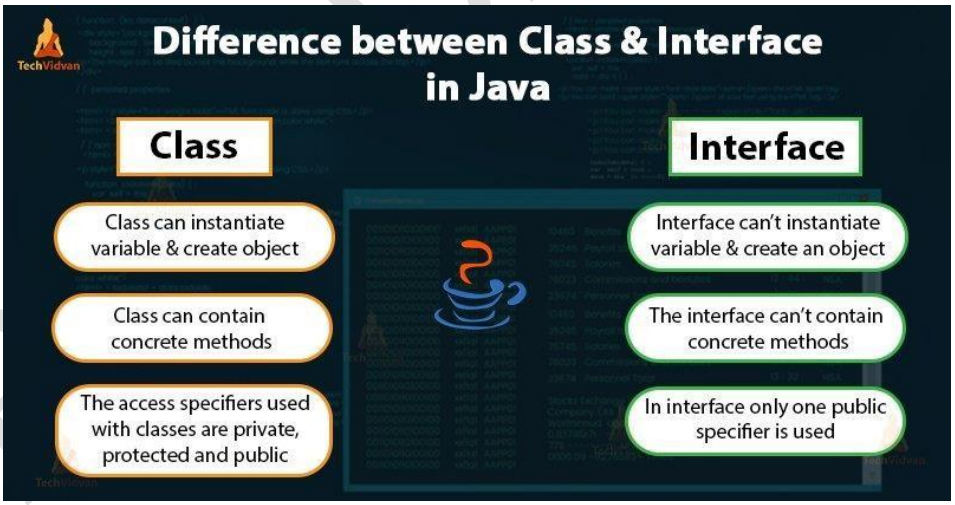


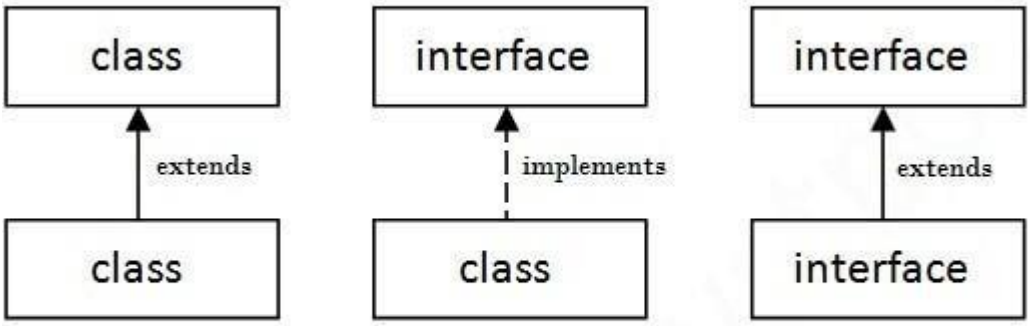
Java: Types of Inheritance

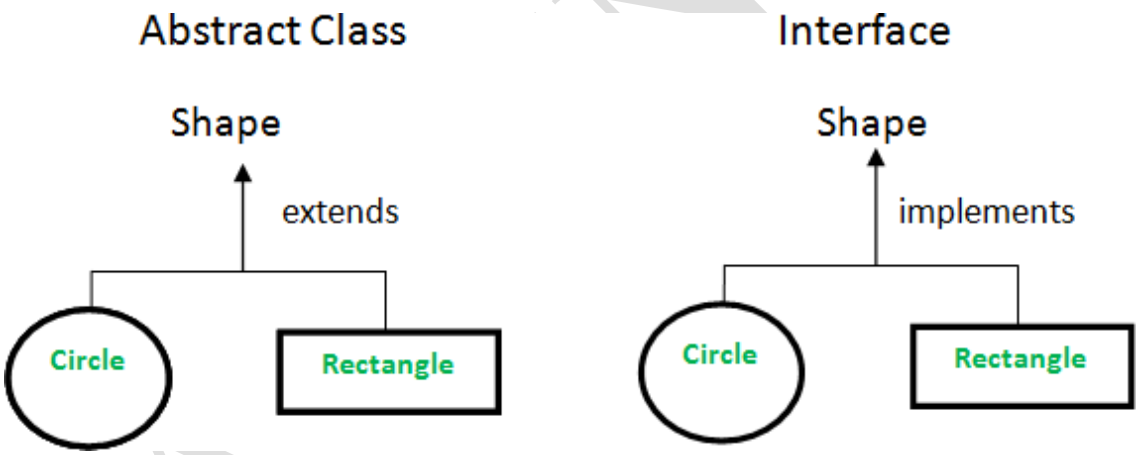
4. Where to use Interfaces in JAVA?

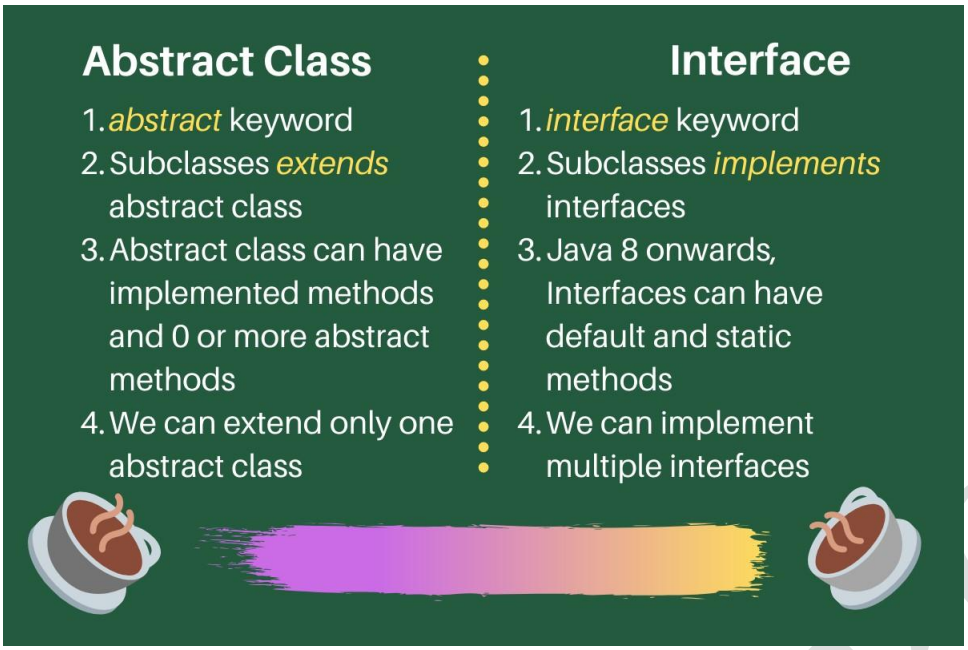

It is used to achieve abstraction.
1
2
By interface, we can support the functionality of multiple inheritance.
It can be used to achieve loose coupling.
3

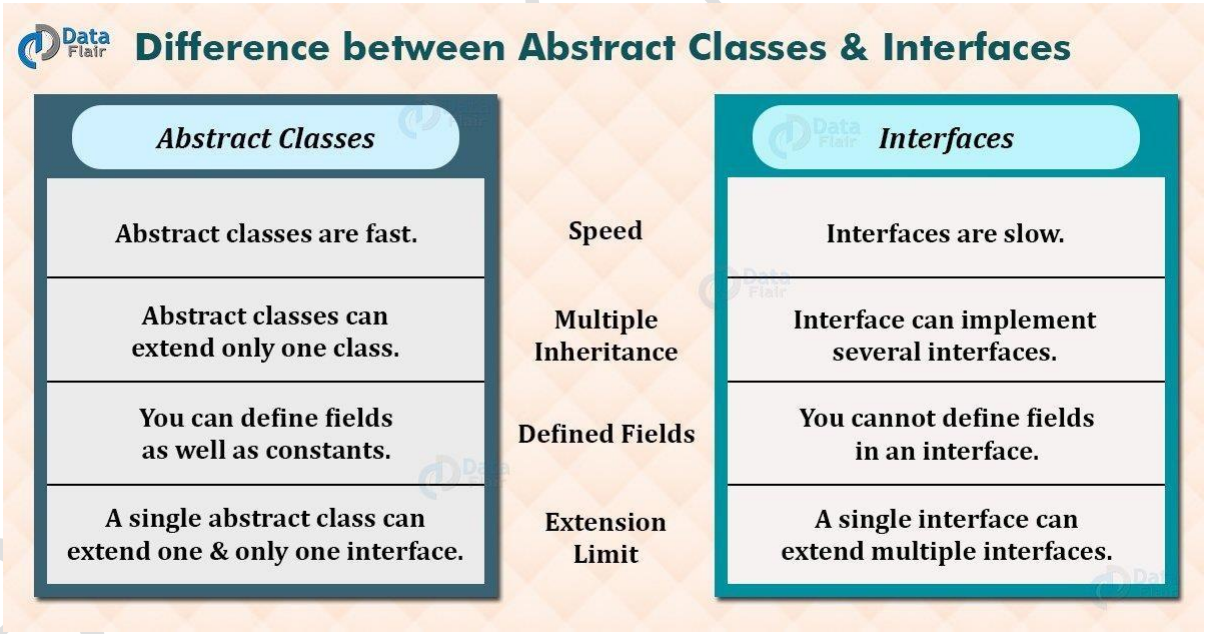

Animal
Pet
DOG
implementing interfaces for entity pet


TechVidvan
Difference between Class & Interface in Java
Class
Interface
Class can instantiate variable & create object
Class can contain concrete methods
The access specifiers used with classes are private, protected and public
Interface can't instantiate variable & create an object
The interface can't contain concrete methods
In interface only one public specifier is used


class
interface
interface
extends
implements
extends
class
class
interface

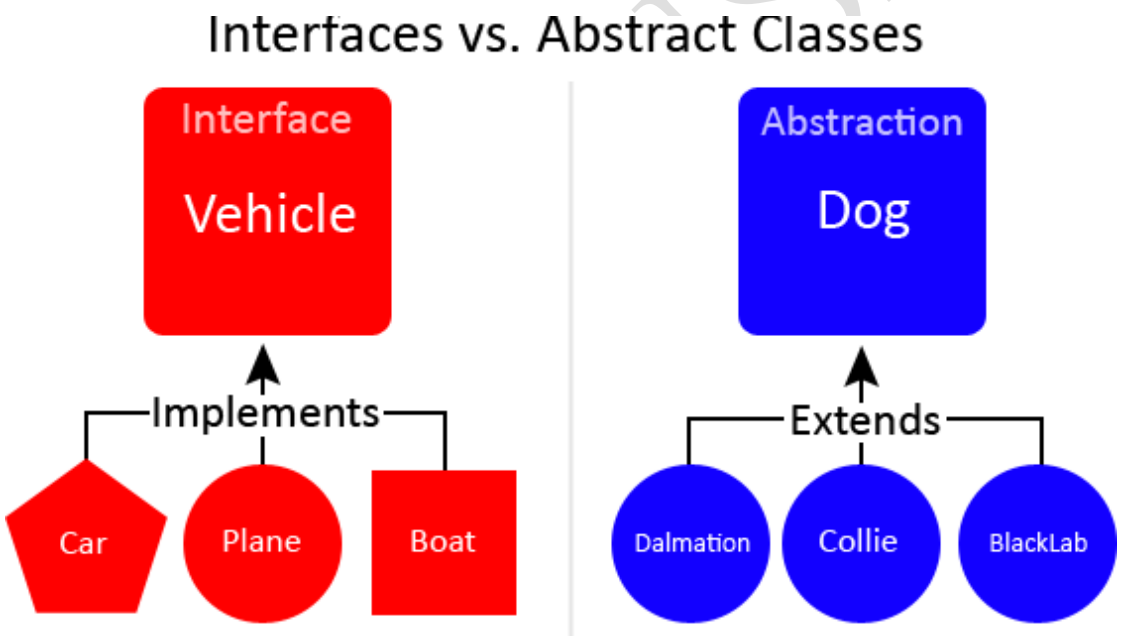

Abstract Class
Shape
extends
Circle
Rectangle
Interface
Shape
implements
Circle
Rectangle


Abstract Class
1. abstract keyword
2. Subclasses extends abstract class
3. Abstract class can have implemented methods and 0 or more abstract methods
4. We can extend only one abstract class
Interface
1. interface keyword
2. Subclasses implements interfaces
3. Java 8 onwards, Interfaces can have default and static methods
4. We can implement multiple interfaces

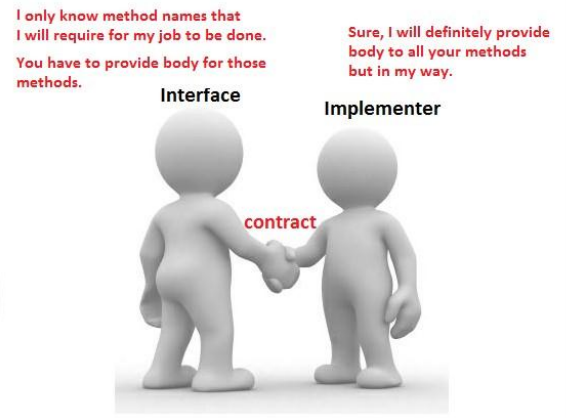

Data Flair
Difference between Abstract Classes & Interfaces
Abstract Classes
Interfaces
Abstract classes are fast.
Speed
Interfaces are slow.
Abstract classes can extend only one class.
Multiple Inheritance
Interface can implement several interfaces.
You can define fields as well as constants.
Defined Fields
You cannot define fields in an interface.
A single abstract class can extend one & only one interface.
Extension Limit
A single interface can extend multiple interfaces.

Abstract Class vs Interface in Java

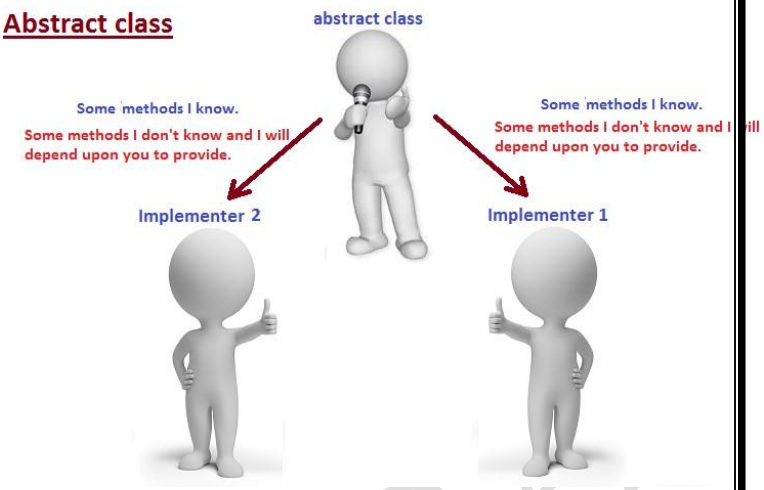
Parameters	Abstract Class	Interface
1. Keyword Used	abstract	interface
2. Type of Variable	Static and Non-static	Static
3. Access Modifiers	All access modifiers	Only public access modifier
4. Speed	Fast	Slow
5. When to use	To avoid Independence	For Future Enhancement

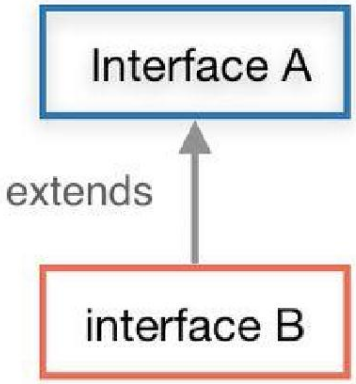


Interface



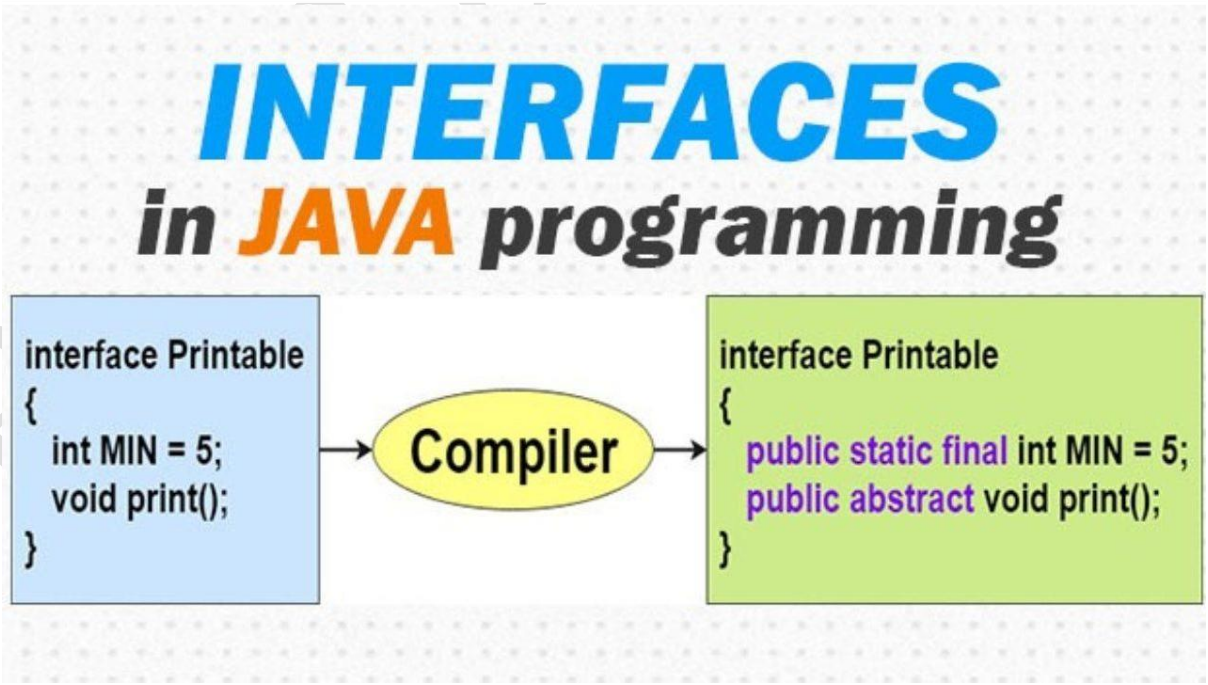
Abstract class






```
public interface A {
        ......
}

public interface B extends A {
        .......
}
```



Example 1: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
Run Code
```

Output

```
The area of the rectangle is 30
```


5. List the Regular Expressions

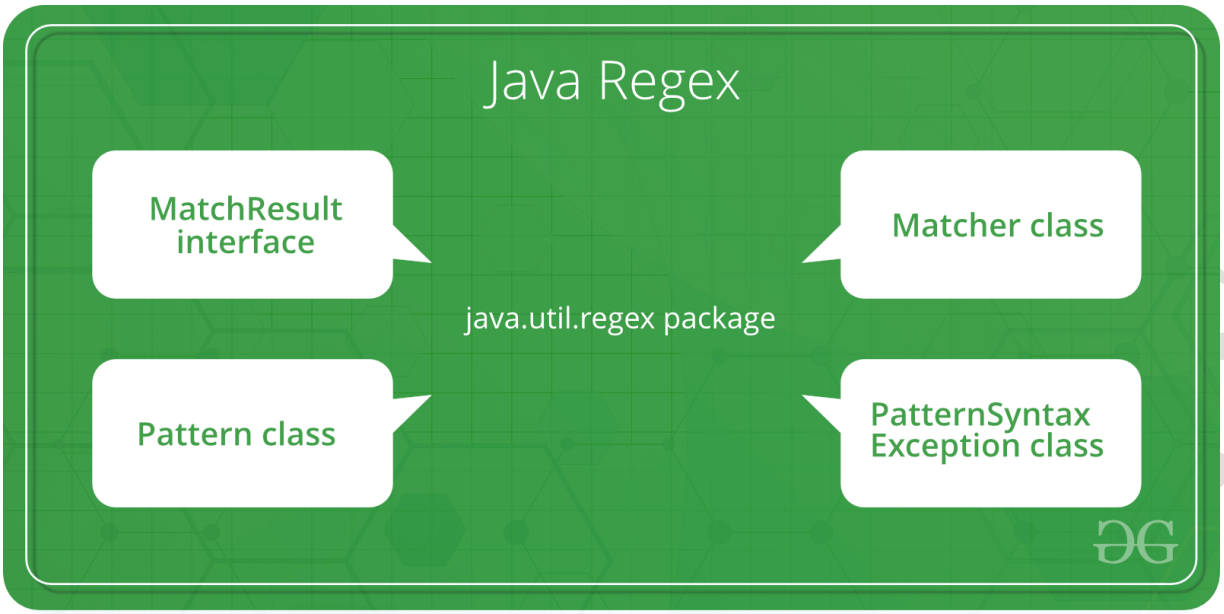
Regular Expressions or Regex (in short) in Java is an API for defining String patterns that can be used for searching, manipulating, and editing a string in Java. Email validation and passwords are a few areas of strings where Regex is widely used to define the constraints. Regular Expressions are provided under **java.util.regex** package. This consists of **3 classes and 1 interface**.

The **java.util.regex** package primarily consists of the following three classes as depicted below in tabular format as follows:

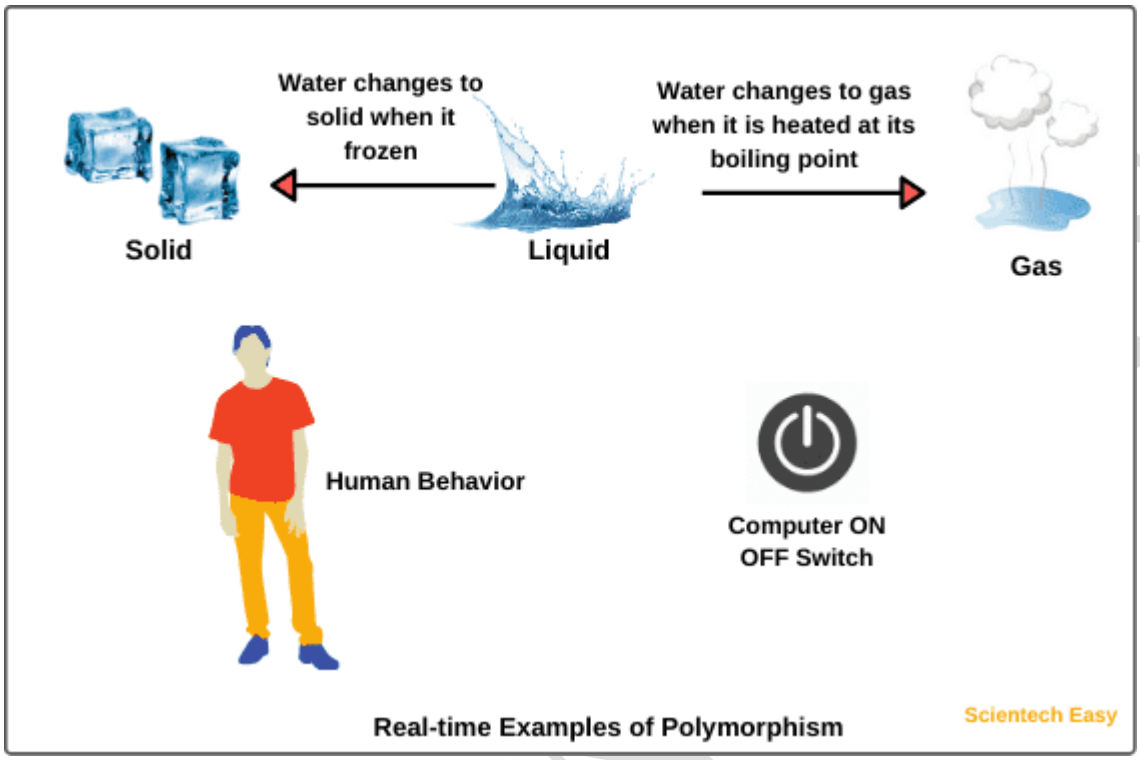
S. No.	Class/Interface	Description
1.	Pattern Class	Used for defining patterns
2.	Matcher Class	Used for performing match operations on text using patterns
3.	PatternSyntaxException Class	Used for indicating syntax error in a regular expression pattern
4.	MatchResult Interface	Used for representing the result of a match operation

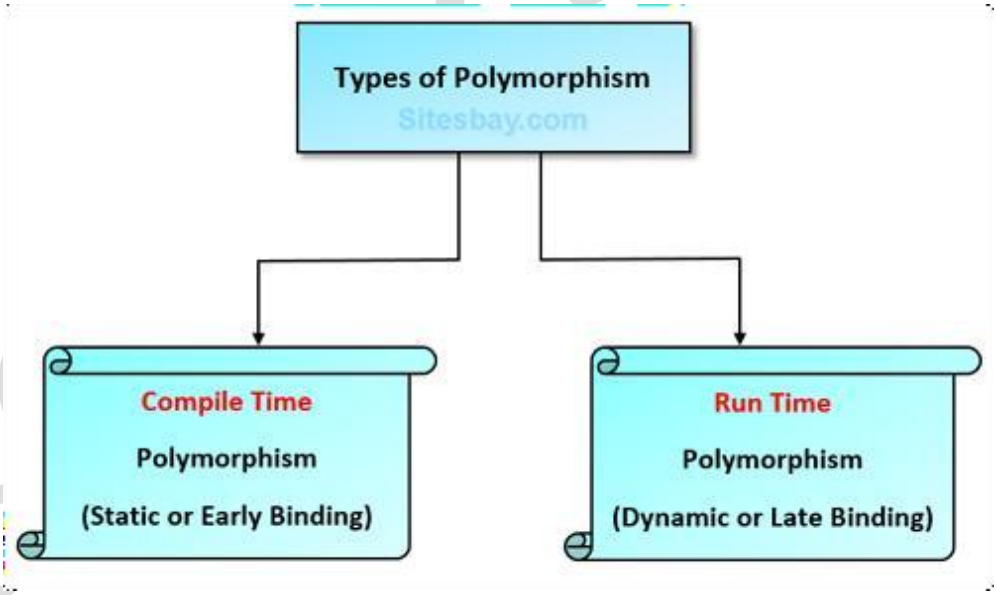
Regex in java provides us with 3 classes and 1 interface listed below as follows:

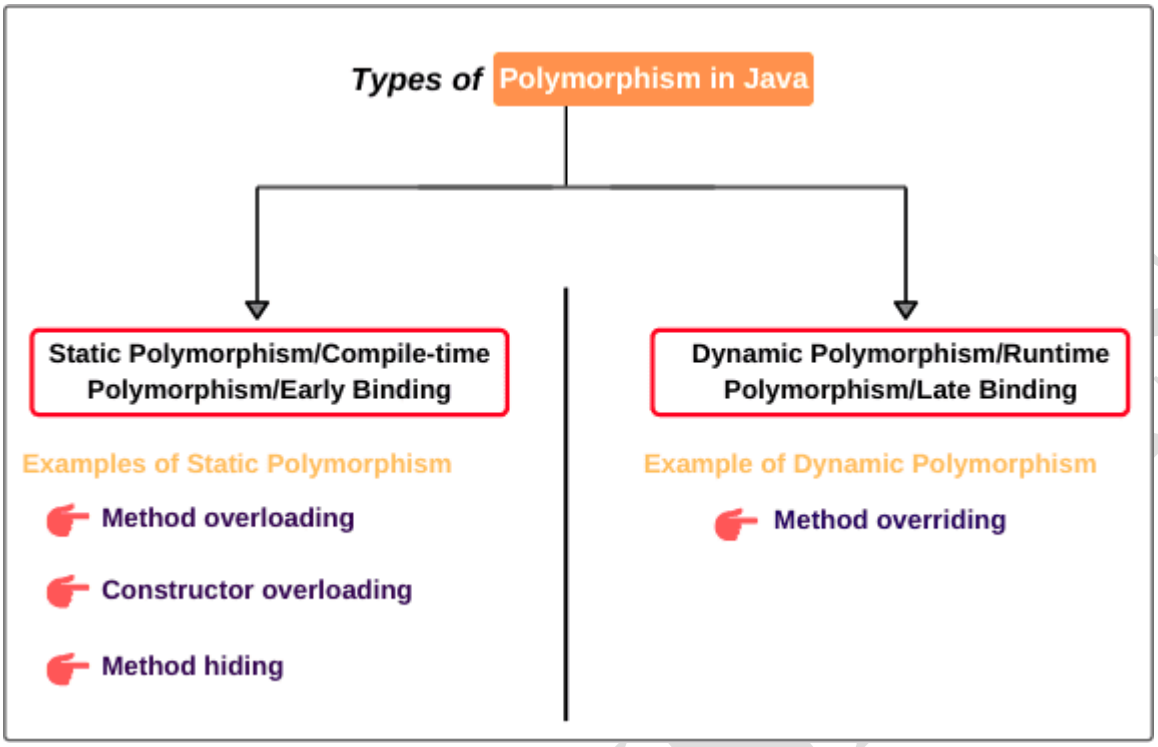
1. Pattern Class
2. Matcher Class
3. PatternSyntaxException Class
4. MatchResult Interface

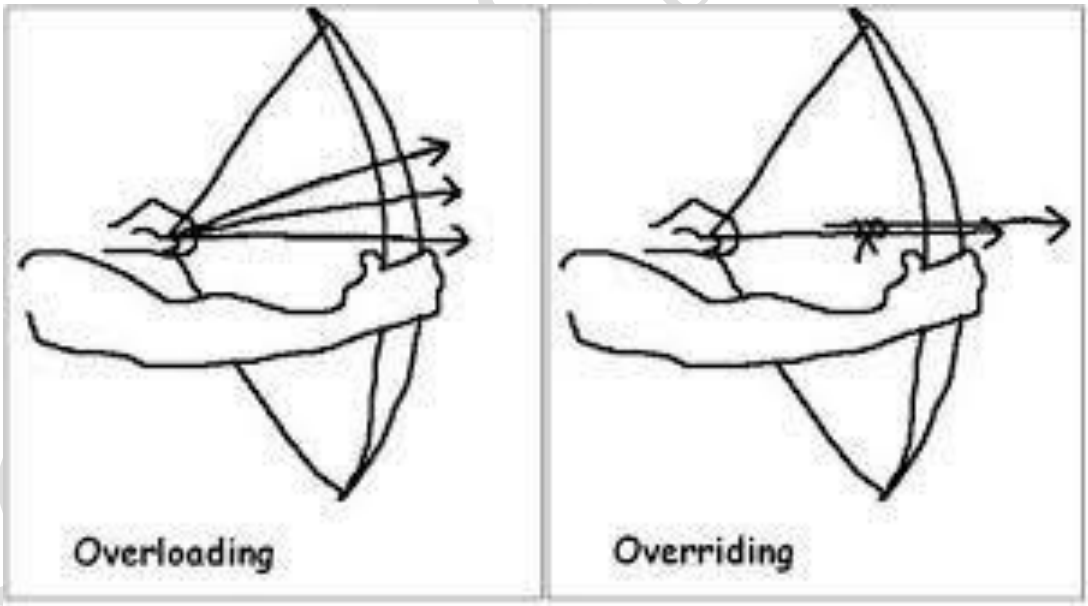


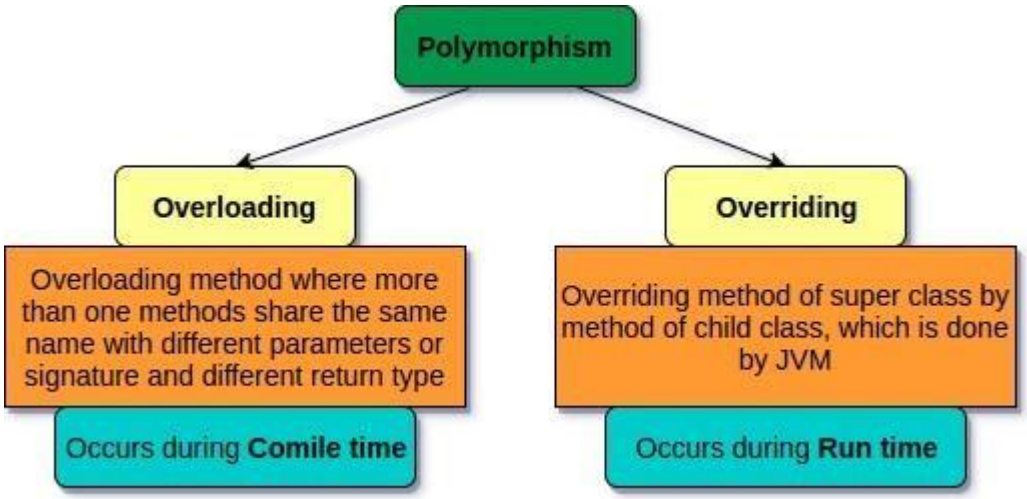
6. Relate on Polymorphism

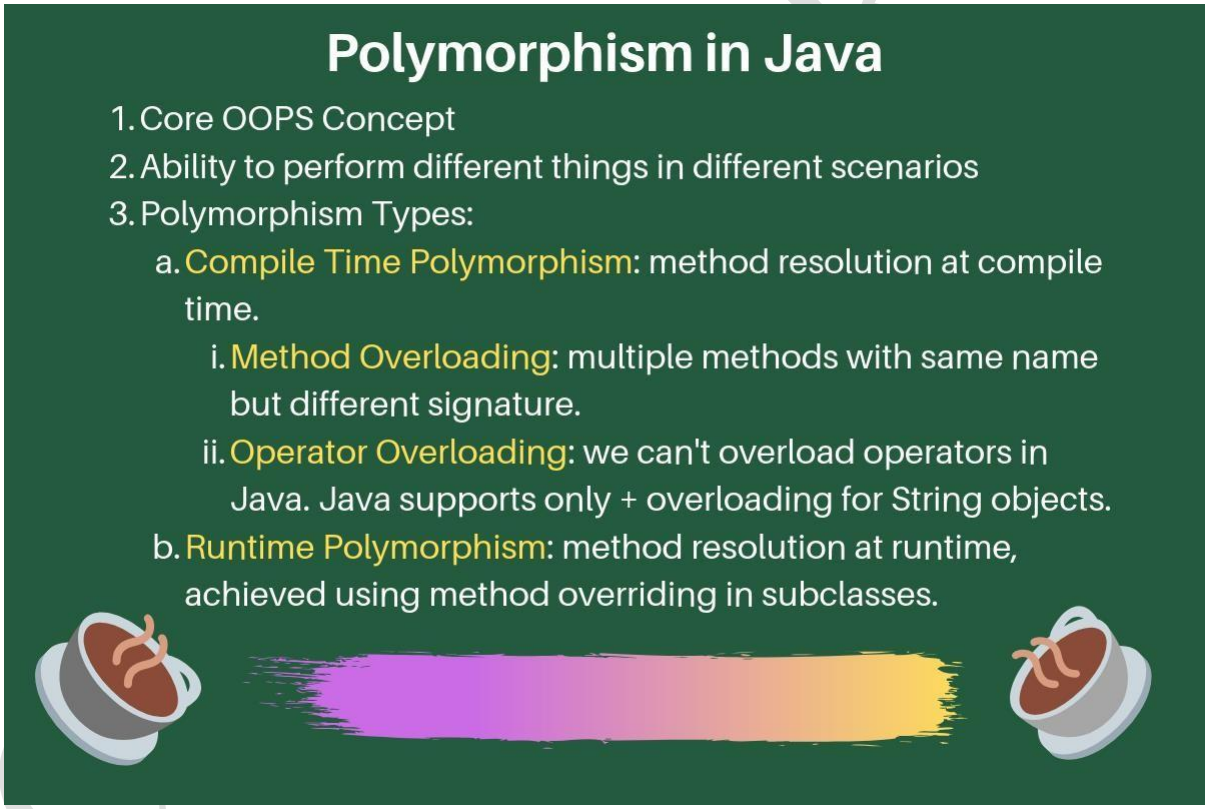

Water changes to solid when it frozen
Water changes to gas when it is heated at its boiling point
Solid
Liquid
Gas
Human Behavior
Computer ON OFF Switch
Real-time Examples of Polymorphism
Scientech Easy


Types of Polymorphism
Sitesbay.com
Compile Time
Polymorphism
(Static or Early Binding)
Run Time
Polymorphism
(Dynamic or Late Binding)


Types of Polymorphism in Java
Static Polymorphism/Compile-time Polymorphism/Early Binding
Dynamic Polymorphism/Runtime Polymorphism/Late Binding
Examples of Static Polymorphism
Method overloading
Constructor overloading
Method hiding
Example of Dynamic Polymorphism
Method overriding


Overloading
Overriding





7. What is Polymorphism and Interfaces in JAVA

Java language is one of the most popular languages among all programming languages. There are several advantages of using the java programming language, whether for security purposes or building large distribution projects. One of the advantages of using JA is that Java tries to connect every concept in the language to the real world with the help of the concepts of classes, inheritance, polymorphism, interfaces, etc. In this article, we will discuss polymorphism and interface concepts.

Polymorphism is that it has many forms that mean one specific defined form is used in many different ways. The simplest real-life example is let's suppose we have to store the name of the person and the phone number of the person, but there are many situations when a person has two different phone numbers. We have to save the same phone number under the same name.

Let us interpret it with help . So, in java, the problem can be solved using an object-oriented concept, *void insertPhone(String name, int phone)*. So, this method is used to save the phone number of the particular person. Similarly, we can use the same form but a different signature means different parameters to store the alternative phone number of the person's **void insertPhone(String name, int phone1, int phone2). One method has two different forms and performs** different **operations.** This is an example of polymorphism, which is method overloading**.**

Types of polymorphism in Java:

1. Run time polymorphism
2. Compile-time polymorphism

Type 1: Run time polymorphism

This type of polymorphism is resolved by the java virtual machine, not by the java compiler. That's why this type of polymorphism is called run-time polymorphism. Run time polymorphism occurs during method overriding in java.

Example

```
import java.io.*;

class GFG1 {

void name() {

    System.out.println("This is the GFG1 class");

  }

}



public class GFG extends GFG1 {



  void name() {

    System.out.println("This is the GFG class");

  }

   public static void main(String[] args) {

    GFG1 ob = new GFG1();

   ob.name();

     GFG1 ob1 = new GFG();
```

```
    ob1.name();

  }

}
```

Output

This is the GFG1 class

This is the GFG class

Output explanation:

In the above example, the same function i.e name is called two times, but in both cases, the output is different. The signatures of these methods are also the same. That's why compilers cannot be able to identify which should be executed. This is determined only after the object creation and reference of the class, which is performed during run time (Memory management). That's why this is run-time polymorphism.

Type 2: Compile-time polymorphism

Method overloading is an example of the compile-time polymorphism method. Overloading means a function having the same name but a different signature. This is compile-time polymorphism because this type of polymorphism is determined during the compilation time because during writing the code we already mention the different types of parameters for the same function name.

Example:

```
import java.io.*;

import java.util.*;

class First {
```

```
    void check()
    {
        System.out.println("This is the class First");
    }
}
 class Second extends First {
    void check(String name)
    {
        System.out.println("This is the class " + name);
    }
     public static void main(String args[])
    {
        Second ob = new Second();
        ob.check("Second");
         First ob1 = new First();
        ob.check();
```

```
            First ob2 = new Second();

        ob.check();

    }

}
```

Output

This is the class Second

This is the class First

This is the class First

Interfaces are very similar to classes. They have variables and methods but the interfaces allow only abstract methods(that don't contain the body of the methods), but what is the difference between the classes and the interfaces? The first advantage is to allow interfaces to implement the multiple inheritances in a particular class. The JAVA language doesn't support multiple inheritances if we extend multiple classes in the class, but with the help of the interfaces, multiple inheritances are allowed in Java.

Real-life Example

The real-world example of interfaces is that we have multiple classes for different levels of employees working in a particular company and the necessary property of the class is the salary of the employees and this. We must be implemented in every class and. Also, it is different for every employee here. The concept of the interface is used. We simply create an interface containing an abstract salary method and implement it in all the classes and we can easily define different salaries of the employees.

Example:

```
interface salary {
    void insertsalary(int salary);
}
class SDE1 implements salary {
    int salary;
    @Override public void insertsalary(int salary)
    {
        this.salary = salary;
    }
    void printSalary() { System.out.println(this.salary); }
}
class SDE2 implements salary {
    int salary;
    @Override public void insertsalary(int salary)
    {
        this.salary = salary;
```

```
    }
    void printSalary() { System.out.println(this.salary); }
}
public class GFG {
    public static void main(String[] args)
    {
        SDE1 ob = new SDE1();
        ob.insertsalary(100000);
        ob.printSalary();
        SDE2 ob1 = new SDE2();
        ob1.insertsalary(200000);
        ob1.printSalary();
    }
}
```

Output

100000

200000

8. Differentiate Classes and Objects in C++ and JAVA

Difference between Class and Object in C++

Class	Object
Class is a container which collection of variables and methods.	object is a instance of class
No memory is allocated at the time of declaration	Sufficient memory space will be allocated for all variables of class at the time of declaration.
One class definition should exist only once in the program.	For one class multiple objects can be created.

9. Define the Inheritance and its types

Inheritance can be implemented in 5 ways and below are types that comes from **OOPS concepts.**

1) Single Inheritance
2) Multi-Level Inheritance
3) Multiple Inheritance
4) Hierarchical Inheritance
5) Hybrid Inheritance

Single Inheritance:

This is very easy to understand. **If a class extends only one class then it is called as Single Inheritance**. Look at the below diagram where class B extends class

A.

Here always only one base class and one child class.

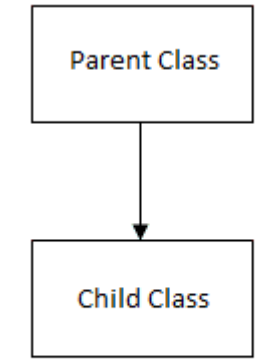


Fig: Single inheritance

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

  A a = new A();
  a.printA();

  B b = new B();
  b.printA();
  b.printB();
 }
}

class A {
 public void printA() {
  System.out.println("PrintA method.");
 }
}

class B extends A {
 public void printB() {
  System.out.println("PrintB method.");
 }
}
```

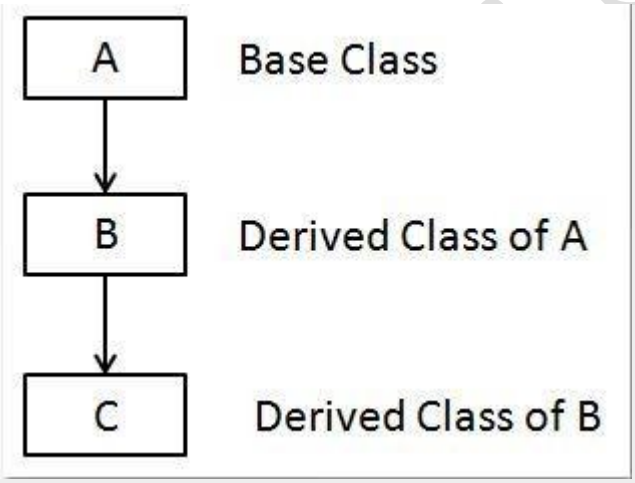
Output:

```
PrintA  method.
PrintA  method.
PrintB method.
```

Through object A, we can call only class A methods. But, both methods can be invoked using B object. Because **class B is inheriting class A properties and methods**.

Multi-Level Inheritance:

If a class is derived from another child or derived class is said to be "Multi-Level Inheritance". Derived class means a child class. This can be minimum of three or more classes involve in this type of inheritance. For example, class B inherits class A and class C inherits class B.




```
package blog.java.w3schools.inheritance;
```

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

  C c = new C();
  c.printA();
  c.printB();
  c.printC();
 }
}

class A {
 public void printA() {
  System.out.println("PrintA method.");
 }
}

class B extends A {
 public void printB() {
  System.out.println("PrintB method.");
 }
}

class C extends B {
 public void printC() {
  System.out.println("PrintC method.");
 }
}
```

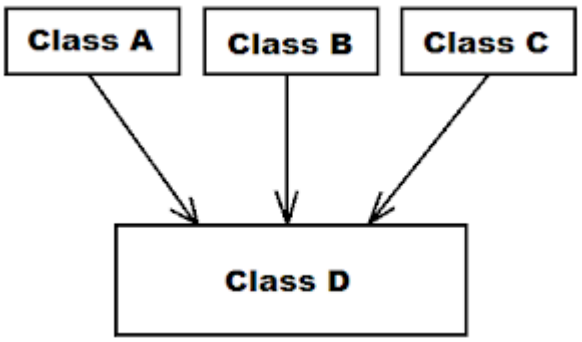
Output:

```
PrintA method.
PrintB method.
PrintC method.
```

Observe carefully, all three methods can be invoked by object c. This is the power of inheritance.

4.3 Multiple Inheritance:

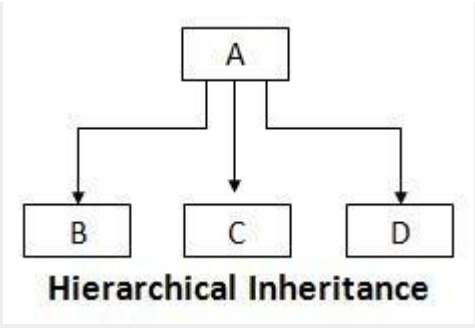
Multiple Inheritance is when one class inherits multiple classes at a time. Basically, **Java does not support Multiple Inheritance**. This is not much implemented any real time projects and not supported in many OO programming languages such as Small Talk, Java, C# do not support Multiple inheritance. Only one language supports Multiple Inheritance is C++.



The main problem is that if both parent classes have a same method then child class which method of parent class will be accessed. This makes ambiguity to the compiler and leads to diamond problem. To avoid all these circumstances, java not supporting it.

4.4 Hierarchical Inheritance:

In simple terms, one base class - multiple child classes. A base class is inherited by many classes(child). Take a look at the below diagram.



Class A is a base class.
Class B inherits class A

Class C inherits class A
Class D inherits class A

class B, C, D are inherited class A.

```
package blog.java.w3schools.inheritance;

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

  B b = new B();
  b.printA();
  b.printB();

  System.out.println(" ----------");

  C c = new C();
  c.printA();
  c.printC();

  System.out.println(" ----------");

  D d = new D();
```

```
  c.printA();
  c.printC();
 }
}

class A {
 public void printA() {
  System.out.println("PrintA method.");
 }
}

class B extends A {
 public void printB() {
  System.out.println("PrintB method.");
 }
}

class C extends A {
 public void printC() {
  System.out.println("PrintC method.");
 }
}

class D extends A {
 public void printD() {
  System.out.println("PrintD method.");
 }
}
```

Output:

```
PrintA method.
PrintB method.
_________-
PrintA method.
PrintC method.
_________-
PrintA method.
```

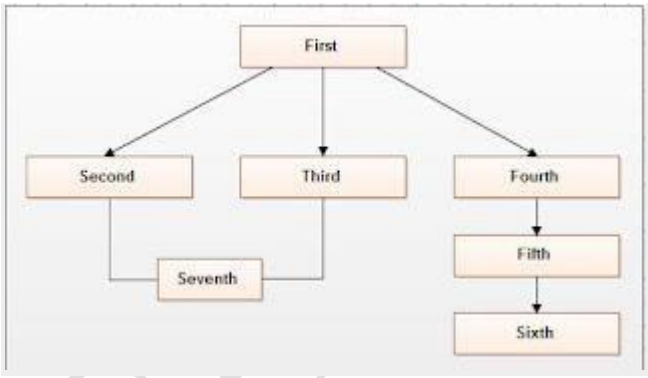
PrintC method.

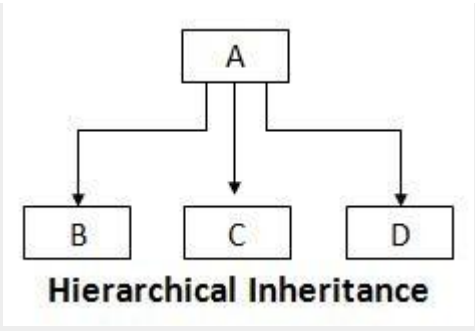
Class B, C, D objects are able to call class A method directly without creating object for class A.

4.5 Hybrid Inheritance:

Hybrid Inheritance is combination of any two or more inheritances together. It may be single and multi level inheritance or multi-level or Hierarchical inheritance.

In this combination, we should **not use multiple inheritance which is not supported by Java**.





10. What is Packages in JAVA

Java Packages

- What is a package?
- **Definition:** A *package* is a grouping of related types providing access protection and name space management. Note that *types* refers to classes, interfaces, enumerations, and annotation types. (Java's Definition)

Package in Java

- Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.
- A package is a collection of related Java entities (such as classes, interfaces, exceptions, errors and enums).
- A **package** provides a mechanism for grouping a variety of similar types of classes, interfaces and sub-packages.
- Grouping is based on functionality.
- Java packages can be stored in compressed files called JAR files (Java Archieve)

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

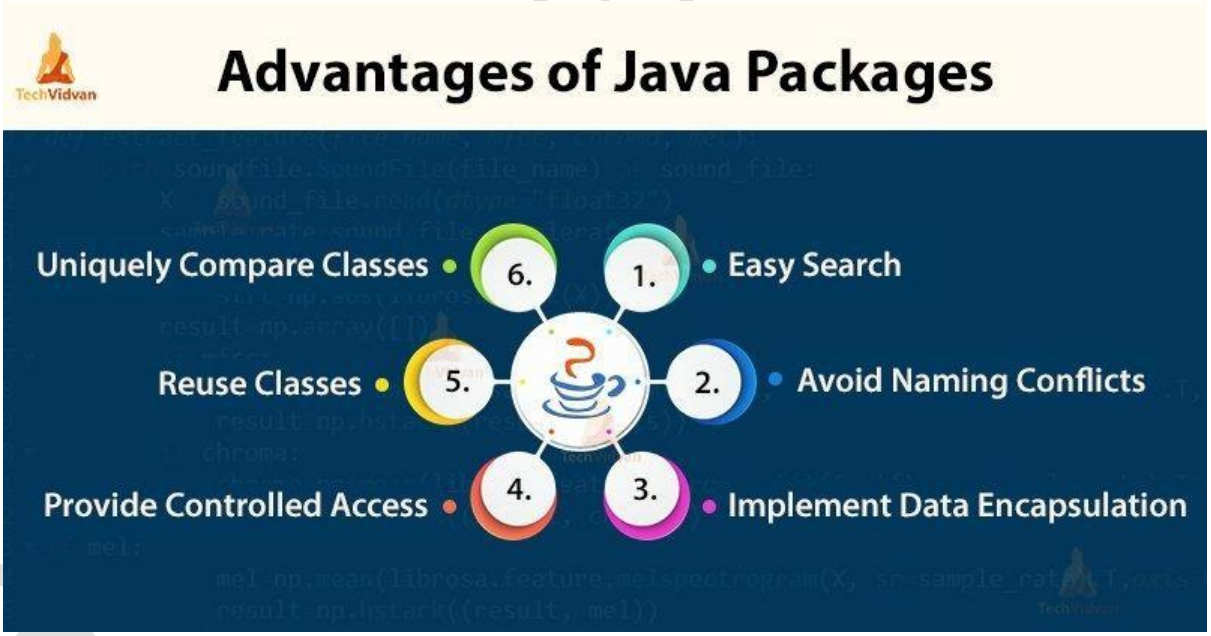
Here, we will have the detailed learning of creating and using user-defined packages.

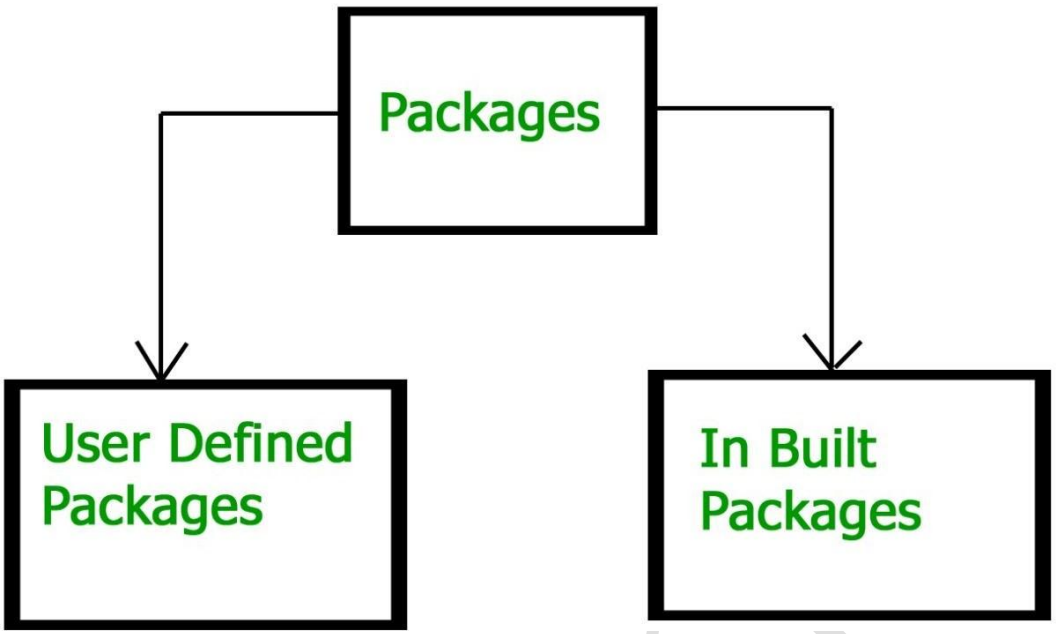
Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

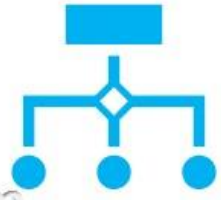




Resolve naming conflict

Reuse of code

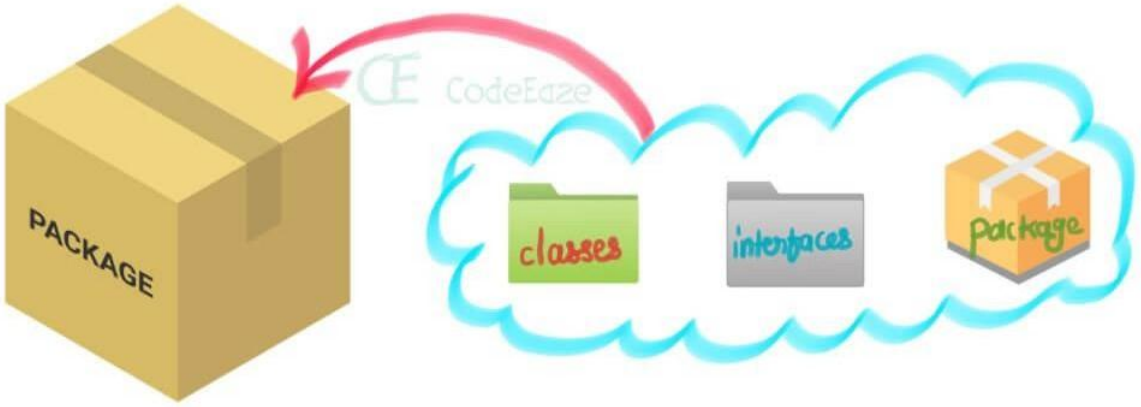
Organization of project



Modularity

Access protection

Information hiding



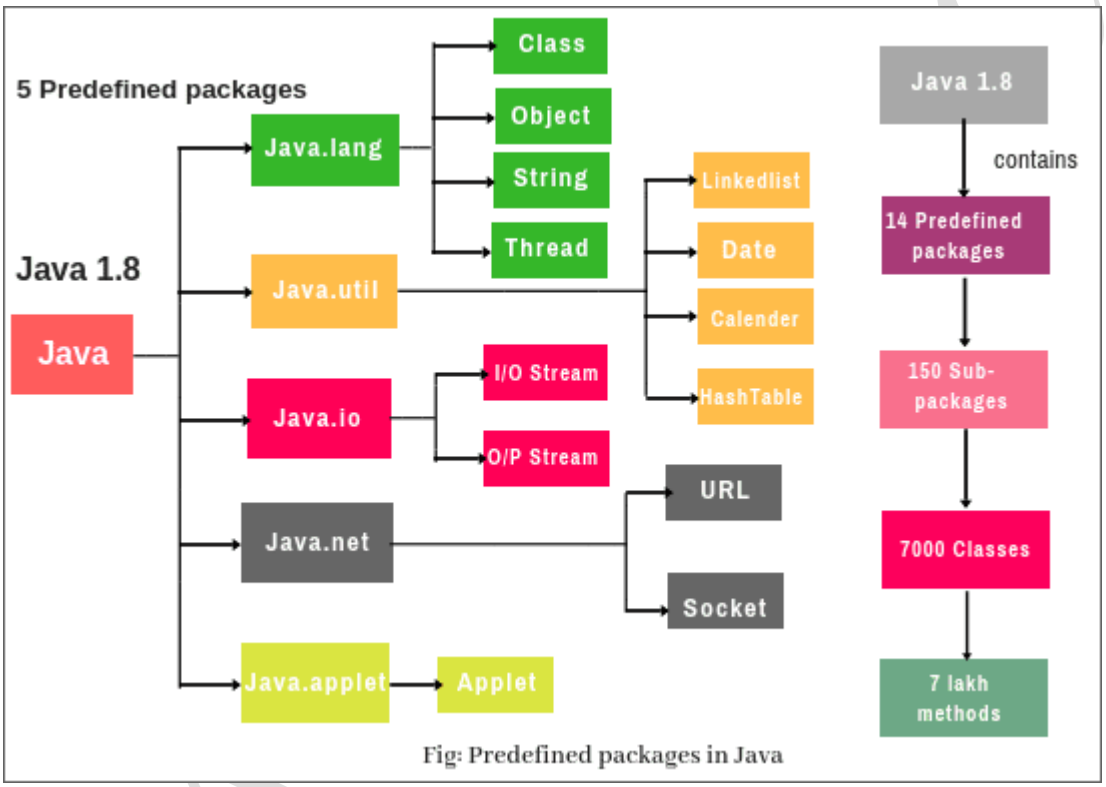


Fig: Predefined packages in Java

11. Discuss in detail the Exception Handling.

Java Exception Handling

In the tutorial, we will learn about different approaches of exception handling in Java with the help of examples.

In the last tutorial, we learned about Java exceptions. We know that exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

- try...catch block
- finally block
- throw and throws keyword

---

1. Java try...catch block

The try-catch block is used to handle exceptions in Java. Here's the syntax of `try...catch` block:

```
try {
  // code
}
catch(Exception e) {
  // code
}
```

Here, we have placed the code that might generate an exception inside the `try` block. Every `try` block is followed by a `catch` block.

When an exception occurs, it is caught by the `catch` block. The `catch` block cannot be used without the `try` block.

Example: Exception handling using try...catch

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

```
    try {

      // code that generate exception
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }
  }
}
```

Run Code

Output

```
ArithmeticException => / by zero
```

In the example, we are trying to divide a number by `0`. Here, this code generates an exception.

To handle the exception, we have put the code, `5 / 0` inside the `try` block. Now when an exception occurs, the rest of the code inside the `try` block is skipped.

The `catch` block catches the exception and statements inside the catch block is executed.

If none of the statements in the `try` block generates an exception, the `catch` block is skipped.

---

2. Java finally block

In Java, the `finally` block is always executed no matter whether there is an exception or not.

The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

The basic syntax of `finally` block is:

```
try {
  //code
}
catch (ExceptionType1 e1) {
  // catch block
}
finally {
  // finally block always executes
}
```

If an exception occurs, the `finally` block is executed after the `try...catch` block. Otherwise, it is executed after the try block. For each `try` block, there can be only one `finally` block.

Example: Java Exception Handling using finally block

```
class Main {
  public static void main(String[] args) {
    try {
      // code that generates exception
```

```
    int divideByZero = 5 / 0;
   }

   catch (ArithmeticException e) {
    System.out.println("ArithmeticException => " + e.getMessage());
   }

   finally {
    System.out.println("This is the finally block");
   }
  }
}
```

Run Code

Output

```
ArithmeticException => / by zero
This is the finally block
```

In the above example, we are dividing a number by **0** inside the `try` block. Here, this code generates an `ArithmeticException`.

The exception is caught by the `catch` block. And, then the `finally` block is executed.

Note: It is a good practice to use the `finally` block. It is because it can include important cleanup codes like,

- code that might be accidentally skipped by return, continue or break
- closing a file or connection

3. Java throw and throws keyword

The Java `throw` keyword is used to explicitly throw a single exception.

When we `throw` an exception, the flow of the program moves from the `try` block to the `catch` block.

Example: Exception handling using Java throw

```
class Main {
  public static void divideByZero() {

    // throw an exception
    throw new ArithmeticException("Trying to divide by 0");
  }

  public static void main(String[] args) {
    divideByZero();
  }
}
```

Run Code

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
        at Main.divideByZero(Main.java:5)
        at Main.main(Main.java:9)
```

In the above example, we are explicitly throwing the `ArithmeticException` using the `throw` keyword.

Similarly, the `throws` keyword is used to declare the type of exceptions that might occur within the method. It is used in the method declaration.

Example: Java throws keyword

```
import java.io.*;

class Main {
  // declareing the type of exception
  public static void findFile() throws IOException {

    // code that may generate IOException
    File newFile = new File("test.txt");
    FileInputStream stream = new FileInputStream(newFile);
  }

  public static void main(String[] args) {
    try {
      findFile();
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

Run Code

Output

```
java.io.FileNotFoundException: test.txt (The system cannot find the file specified)
```

When we run this program, if the file **test.txt** does not exist, FileInputStream throws a FileNotFoundException which extends the IOException class.

The findFile() method specifies that an IOException can be thrown. The main() method calls this method and handles the exception if it is thrown.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the throws clause.

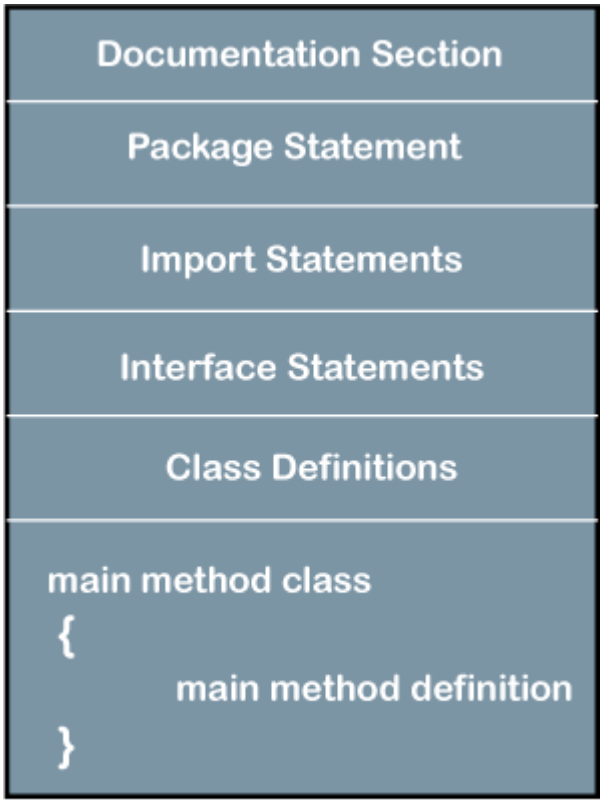
12.Summarize the JAVA Structure

Structure of Java Program

Java is an object-oriented programming

, **platform-independent,** and **secure** programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications. So, before diving in depth, it is necessary to understand the **basic structure of Java program** in detail. In this section, we have discussed the basic **structure of a Java program**. At the end of this section, you will able to develop the Hello world Java program

, easily.



Structure of Java Program

Let's see which elements are included in the structure of a Java program

. A typical structure of a Java

program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables
- Main Method Class
- Methods and Behaviors

Documentation Section

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line,** and **documentation** comments.

- **Single-line Comment:** It starts with a pair of forwarding slash **(//)**. For example:

```
1. //First Java Program
```

- **Multi-line Comment:** It starts with a **/*** and ends with ***/.** We write between these two symbols. For example:

```
1. /*It is an example of
2. multiline comment*/
```

- **Documentation Comment:** It starts with the delimiter **(/**)** and ends with ***/**. For example:

```
1. /**It is an example of documentation comment*/
```

Package Declaration

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** statement in a Java program. It must be defined before any class and interface declaration. It is necessary because a Java

class can be placed in different packages and directories based on the module they are used. For all these classes package belongs to a single parent directory. We use the keyword **package** to declare the package name. For example:

Competitive questions on Structures in HindiKeep Watching

1. **package** javatpoint; //where javatpoint is the package name
2. **package** com.javatpoint; //where com is the root directory and javatpoint is the subdirectory

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the **import** keyword to import the class. It is written before the class declaration and after the package statement. We use the import statement in two ways, either import a specific class or import all classes of a particular package. In a Java program, we can use multiple import statements. For example:

1. **import** java.util.Scanner; //it imports the Scanner class only
2. **import** java.util.*; //it imports all the class of the java.util package

Interface Section

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface

is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword. An interface can also be used with other interfaces by using the **extends** keyword. For example:

```
1. interface car
2. {
3.  void start();
4.  void stop();
5. }
```

Class Definition

In this section, we define the class. It is **vital** part of a Java program. Without the class

, we cannot create any Java program. A Java program may conation more than one class definition. We use the **class** keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method. For example:

```
1. class Student //class definition
2. {
3.  }
```

Class Variables and Constants

In this section, we define variables

and **constants** that are to be used later in the program. In a Java program, the variables and constants are defined just after the class definition. The variables

and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It defines the life of the variables. For example:

```
1. class Student //class definition
2. {
3.  String sname;  //variable
4.  int id;
5.  double percentage;
6. }
```

Main Method Class

In this section, we define the **main() method.** It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the main() method:

```
1. public static void main(String args[])
2. {
3.  }
```

For example:

```
1. public class Student //class definition
2. {
3. public static void main(String args[])
4. {
5. //statements
6. }
7.  }
```

You can read more about the Java main() method here

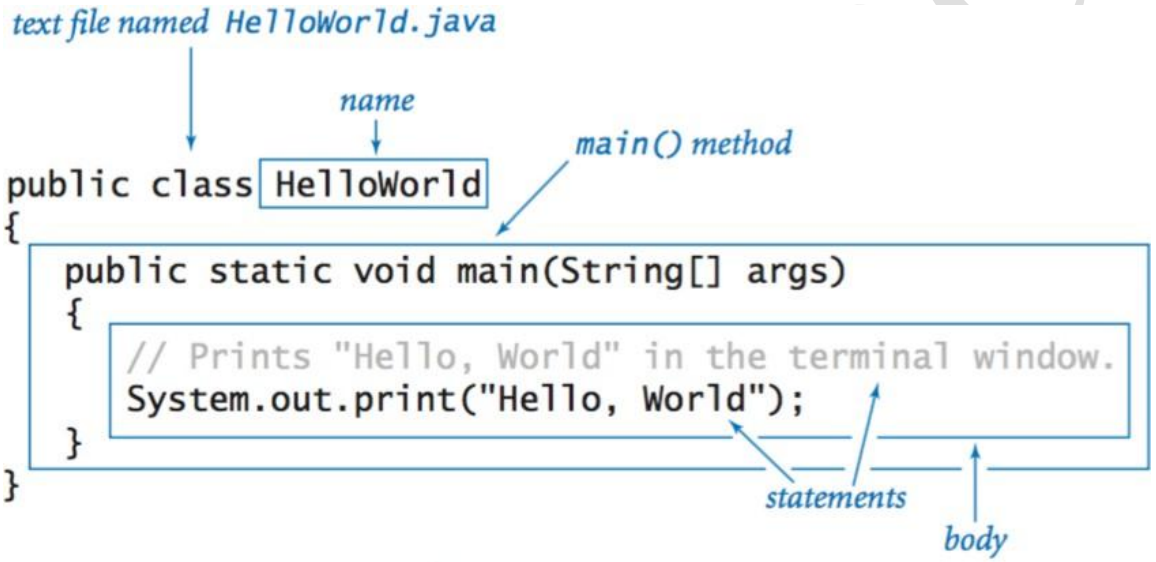
Methods and behavior

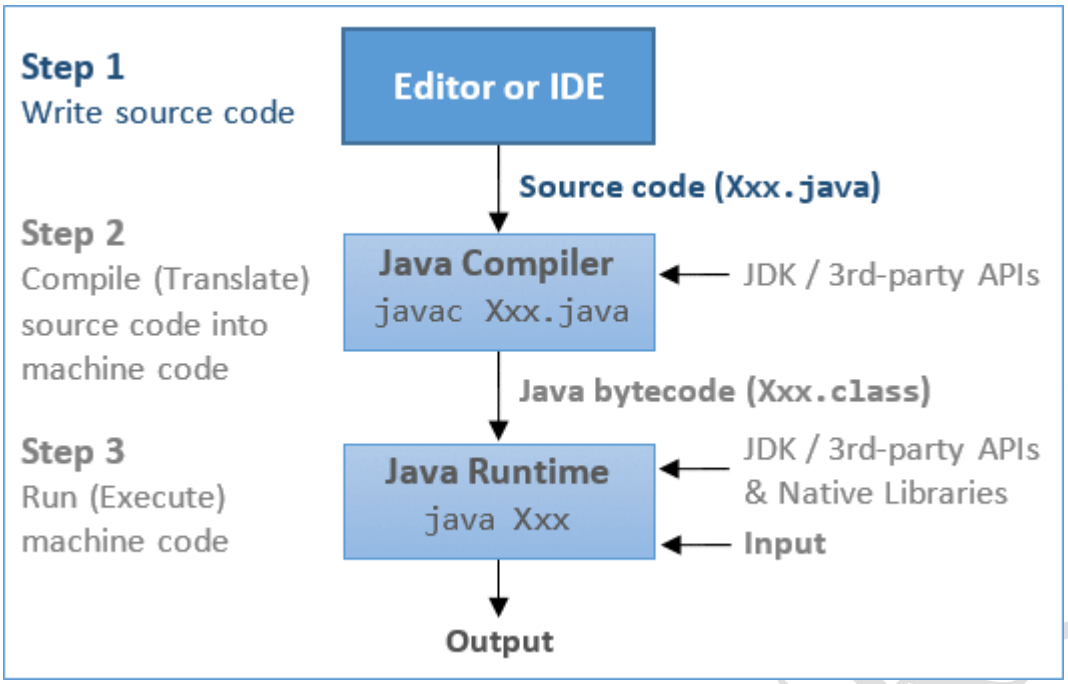
In this section, we define the functionality of the program by using the methods

. The methods are the set of instructions that we want to perform. These instructions execute at runtime and perform the specified task. For example:

```
1. public class Demo //class definition
2. {
3. public static void main(String args[])
4. {
5. void display()
6. {
7. System.out.println("Welcome to java");
8. }
9.  //statements
10.}
11.}
```

```
package details                    ------------>    import java.io.*
class className                    ------------>    class Sum
{
Data members;                      ------------>    int a, b, c;
user_defined method;               ------------>    void display();
public static void main(String args[])
{
Block of Statements;               ------------>    System.out.println("Hello Java !");
}
}
```



13. How to add Interfaces to your JAVA Program?

In the last tutorial we discussed abstract class which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction. Abstraction is a process where you show only "relevant" data and "hide" unnecessary details of an object from the user(See: Abstraction). In this guide, we will cover **what is an interface in java**, why we use it and what are rules that we must follow while using interfaces in Java Programming.

What is an interface in Java?

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: Java abstract method). Also, the variables declared in an interface are public, static & final by default. We will cover this in detail, later in this guide.

What is the use of interface in Java?

As mentioned above they are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

Syntax:

Interfaces are declared by specifying a keyword "interface". E.g.:

```
interface MyInterface
{
  /* All the methods are public abstract by default
   * As you see they have no body
   */
  public void method1();
  public void method2();
}
```

Example of an Interface in Java

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

Do you know? class implements interface but an interface extends another interface.

```
interface MyInterface
```

```
{
  /* compiler will treat them as:
   * public abstract void method1();
   * public abstract void method2();
   */
  public void method1();
  public void method2();
}
class Demo implements MyInterface
{
  /* This class must have to implement both the abstract methods
   * else you will get compilation error
   */
  public void method1()
  {
        System.out.println("implementation of method1");
  }
  public void method2()
  {
        System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
        MyInterface obj = new Demo();
        obj.method1();
  }
}
```

Output:

implementation of method1

14. Describe about designing tools in multimedia

15. Discuss about Exception Handling in JAVA

Java Exception Handling

In the tutorial, we will learn about different approaches of exception handling in Java with the help of examples.

In the last tutorial, we learned about Java exceptions. We know that exceptions abnormally terminate the execution of a program.
This is why it is important to handle exceptions. Here's a list of different approaches to handle exceptions in Java.

- try...catch block
- finally block
- throw and throws keyword

---

1. Java try...catch block

The try-catch block is used to handle exceptions in Java. Here's the syntax of try...catch block:

```
try {
  // code
}
catch(Exception e) {
  // code
}
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by a catch block.

When an exception occurs, it is caught by the catch block. The catch block cannot be used without the try block.

Example: Exception handling using try...catch

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

    try {

      // code that generate exception
      int divideByZero = 5 / 0;
      System.out.println("Rest of code in try block");
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }
  }
```

```
}
```
Run Code

Output

```
ArithmeticException => / by zero
```

In the example, we are trying to divide a number by `0`. Here, this code generates an exception.

To handle the exception, we have put the code, `5 / 0` inside the `try` block. Now when an exception occurs, the rest of the code inside the `try` block is skipped.

The `catch` block catches the exception and statements inside the catch block is executed.

If none of the statements in the `try` block generates an exception, the `catch` block is skipped.

---

2. Java finally block

In Java, the `finally` block is always executed no matter whether there is an exception or not.

The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

The basic syntax of `finally` block is:

```
try {
  //code
}
catch (ExceptionType1 e1) {
```

```
  // catch block
}
finally {
  // finally block always executes
}
```

If an exception occurs, the `finally` block is executed after the `try...catch` block. Otherwise, it is executed after the try block. For each `try` block, there can be only one `finally` block.

Example: Java Exception Handling using finally block

```
class Main {
  public static void main(String[] args) {
    try {
      // code that generates exception
      int divideByZero = 5 / 0;
    }

    catch (ArithmeticException e) {
      System.out.println("ArithmeticException => " + e.getMessage());
    }

    finally {
      System.out.println("This is the finally block");
    }
  }
```

```
}
```
Run Code

Output

```
ArithmeticException => / by zero
This is the finally block
```

In the above example, we are dividing a number by **0** inside the `try` block. Here, this code generates an `ArithmeticException`.

The exception is caught by the `catch` block. And, then the `finally` block is executed.

Note: It is a good practice to use the `finally` block. It is because it can include important cleanup codes like,

- code that might be accidentally skipped by return, continue or break
- closing a file or connection

---

3. Java throw and throws keyword

The Java `throw` keyword is used to explicitly throw a single exception.

When we `throw` an exception, the flow of the program moves from the `try` block to the `catch` block.

Example: Exception handling using Java throw

```
class Main {
  public static void divideByZero() {

    // throw an exception
```

```
    throw new ArithmeticException("Trying to divide by 0");
  }

  public static void main(String[] args) {
    divideByZero();
  }
}
```

Run Code

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
        at Main.divideByZero(Main.java:5)
        at Main.main(Main.java:9)
```

In the above example, we are explicitly throwing the `ArithmeticException` using the `throw` keyword.

Similarly, the `throws` keyword is used to declare the type of exceptions that might occur within the method. It is used in the method declaration.

Example: Java throws keyword

```
import java.io.*;

class Main {
  // declareing the type of exception
  public static void findFile() throws IOException {

    // code that may generate IOException
    File newFile = new File("test.txt");
    FileInputStream stream = new FileInputStream(newFile);
  }
```

```
  public static void main(String[] args) {
    try {
      findFile();
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

Run Code

Output

```
java.io.FileNotFoundException: test.txt (The system cannot find the file specified)
```

When we run this program, if the file **test.txt** does not exist, `FileInputStream` throws a `FileNotFoundException` which extends the `IOException` class.

The `findFile()` method specifies that an `IOException` can be thrown. The `main()` method calls this method and handles the exception if it is thrown.

If a method does not handle exceptions, the type of exceptions that may occur within it must be specified in the `throws` clause.