**OOPs Concept in Java**

**Object Oriented Programing (OOPs):** It is a methodology to design a program or software solutions using objects and classes.

**Objects:** Real world entities that has their own properties and behaviours (Real world entity means which we can see touch feel it is an object). Every object will have some state and behaviour

**Example:** Laptop, Mobile Phone,…

**Classes:** Blueprint from which an objects properties and behaviours are decided (How object will look like, how going to be represented) it is called class.

**Example:**

Class is dog

Properties: breed, size, age, colour

Behaviour: Eat, sleep, run, Bark

* Properties will same but the value of property will change.

**OOPs vs Procedural Programming:**

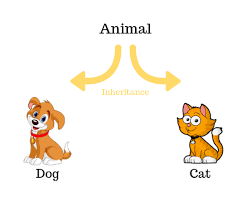
|  |  |
| --- | --- |
| **Object Oriented Programming** | **Procedural Programming** |
| Bottom up approach – think about object first and starting to write the code. | Top down approach – Just start writing the program. |
| Divided into objects | Divided into functions |
| For security we have access modifiers – private, public, protect | Don’t have access modifiers |
| Objects can move & communicate with each other through member functions | Data can move freely from function to function in the system |
| More secure | Less secure |
| Supports overloading | Do not support overloading |

**OOPs Concepts:**

1. Inheritance
2. Polymorphism
3. Abstraction
4. Encapsulation
5. **Inheritance:**

Inheritance is the property of an object to acquire all the properties and behaviour of its parent object. It represents the **IS-A** relationship which is known as a parent-child relationship.

**Example:**

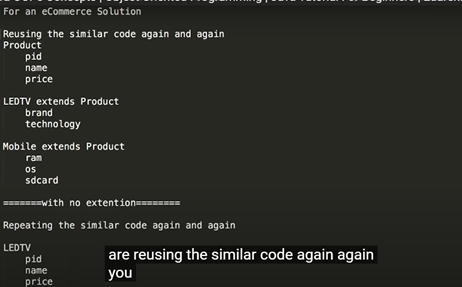


**Example:**

Consider that there is an object structure called product. So product is going to have a product ID, Name, price. These are the few of attributes associated with the product.

**Use case:**

For an E-commerce solution. There will be product which can be LED T. So LED TV will extend the Product. LED TV attributes. What is the brand, technology, any other attributes. Mobile phone extends product.

**Note:** Here extends means inheritance. With no extension reusing the similar code again and again.

**Syntax:**

Class subclass extends super class

{

}

**Advantages:**

1. Code reusability
2. Extensibility
3. Overriding
4. Data hiding

***Types of inheritance :-***

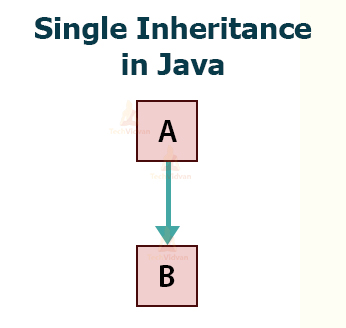
There are five types of inheritance in java,

* 1. ***Single inheritance***
  2. ***Multilevel inheritance***
  3. ***Hierarchical inheritance***
  4. ***Multiple inheritance***
  5. ***Hybrid Inheritance***

Single inheritance:-

In single inheritance, there is a single child class that inherits properties from one parent class.

In the following diagram, class A is a base class that is derived from class B. It is also known as single-level inheritance.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/single-inheritance-in-java.jpg)

Syntax:

class A

{

//methods and fields

}

**Class** B **extends** A

{

//methods and fields

}

***Example1:-***

**class** Faculty

{

**float** salary=30000;

}

**class** Science **extends** Faculty

{

**float** bonous=2000;

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

{

Science obj=**new** Science();

System.**out**.println("Salary is:"+obj.salary);

System.**out**.println("Bonous is:"+obj.bonous);

}

}

**Output:**

Salary is: 30000.0

Bonous is: 2000.0

**Example 2:**

package com.techvidvan.inheritance;

//Base class

**class** Person

{

String name="John";

public **void** show()

{

System.out.println("Student inheriting properties from Person");

}

}

//child class

**class** Student **extends** Person

{

// defining additional properties to child class

String course = "Techvidvan's Java Course";

public **void** show1()

{

System.out.println("I am a Student who belongs to Person class");

}

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

{

Student obj = **new** Student();

obj.show();

obj.show1();

System.out.println("Name of student: " +obj.name);

System.out.println("Course opted by the student: " +obj.course);

}

}

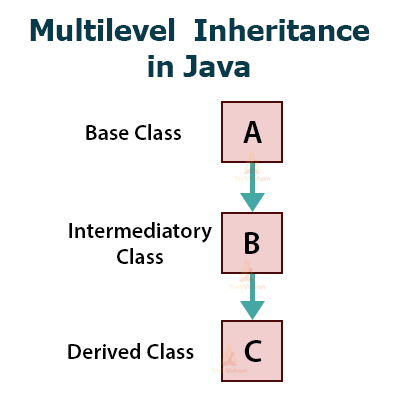
**Output:**

Student inheriting properties from Person  
I am a Student who belongs to Person class  
Name of student: John  
Course opted by the student: Techvidvan’s Java Course

Multilevel inheritance:-

In this type of inheritance, the child or derived class inherits the features of the superclass and simultaneously this child class acts as a superclass for another derived class.

In the following diagram, class A is a base class that is derived from class B, which in turn, acts as a base class for a derived class C.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/multilevel-inheritance-in-java.jpg)

***Example:-***

class A

{ void m1(){System.out.println("m1 method");}

};

class B extends A

{ void m2(){System.out.println("m2 method");}

};

class C extends B

{ void m3(){System.out.println("m3 method");} public static void main(String[] args)

{ A a = new A(); a.m1();

B b = new B(); b.m1(); b.m2();;

C c = new C(); c.m1(); c.m2(); c.m3();

}

}

**Example:**

**class** Faculty

{

**float** total\_sal=0, salary=30000;

}

**class** HRA **extends** Faculty

{

**float** hra=3000;

}

**class** DA **extends** HRA

{

**float** da=2000;

}

**class** Science **extends** DA

{

**float** bonous=2000;

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

{

Science obj=**new** Science();

obj.total\_sal=obj.salary+obj.hra+obj.da+obj.bonous;

System.**out**.println("Total Salary is:"+obj.total\_sal);

}

}

**Output:**

Total Salary is: 37000.0

***Example:***

package com.techvidvan.inheritance;

//Base class

**class** Person

{

public **void** show()

{

System.out.println("Student inheriting properties from Person");

}

}

**class** Student **extends** Person

{

public **void** show1()

{

System.out.println("I am a Student who belongs to Person class");

}

}

//child class

**class** EngineeringStudent **extends** Student

{

// defining additional properties to the child class

public **void** show2()

{

System.out.println("Engineering Student inheriting properties from Student");

}

}

public **class** MultilevelDemo

{

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

{

EngineeringStudent obj = **new** EngineeringStudent();

obj.show();

obj.show1();

obj.show2();

}

}

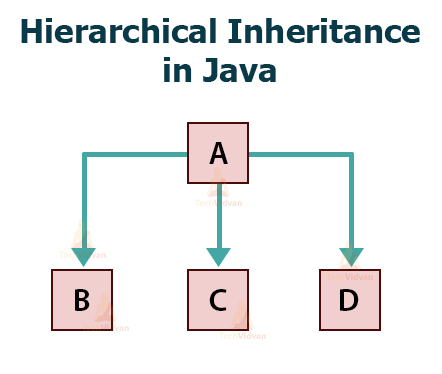
***Output:***

Student inheriting properties from Person  
I am a Student who belongs to Person class  
Engineering Student inheriting properties from Student

***Hierarchical inheritance :-***

In Hierarchical Inheritance, one class acts as a superclass (base class) for more than one subclass. More than one subclass can inherit the features of a base class.

In the following diagram, class A is a base class for the derived classes B, C, and D.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/hierarchial-inheritance-in-java.jpg)

***Example:-***

class A

{ }

class B extends A

{ }

class C extends A

{ }

**Example:**

**class** Person

{

public **void** show()

{

System.out.println("I am a Person");

}

}

//child class1

**class** Student **extends** Person

{

public **void** show1()

{

System.out.println("I am a Student who is Person ");

}

}

//child class2

**class** Teacher **extends** Person

{

// defining additional properties to the child class

public **void** show2()

{

System.out.println("I am a Teacher who is a Person");

}

}

//child class3

**class** Doctor **extends** Person

{

// defining additional properties to the child class

public **void** show3()

{

System.out.println("I am a Doctor who is a Person");

}

}

public **class** HierarchicalInheritance

{

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

{

Teacher teacher = **new** Teacher();

Student student = **new** Student();

Doctor doctor = **new** Doctor();

student.show();

student.show1();

teacher.show2();

doctor.show3();

}

}

**Output:**

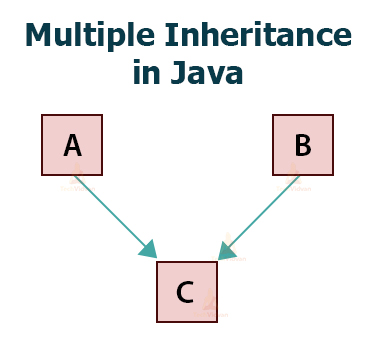
I am a Person  
I am a Student who is Person  
I am a Teacher who is a Person  
I am a Doctor who is a Person

***Multiple inheritance:-***

In Multiple Inheritance, one child or subclass class can have more than one base class or superclass and inherit features from every parent class which it inherits.

We have already discussed that Java does not support multiple inheritances with classes. We can achieve multiple inheritances only with the help of Interfaces.

In the following diagram, Class C inherits from interfaces A and B.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/multiple-inheritance-in-java.jpg)

***Class A extends B ===>valid Class A extends B ,C ===>invalid***

***Example:-***

class Parent1

{ void money() {System.out.println("parent1 money");}

};

class Parent2

{ void money() {System.out.println("parent2 money");}

};

class Child extends Parent1,Parent2

{ public static void main(String[] args)

{ Child c = new Child();

*c.money();* ***//ambiguity problems***

}

};

**Example:**

package com.techvidvan.inheritance;

//base interface1

interface Moveable

{

public **void** run();

}

//base interface2

interface Speakable

{

public **void** speak();

}

//child interface inheriting two base interfaces

interface Ability **extends** Moveable, Speakable

{

public **void** show();

}

**class** Person **implements** Ability

{

@Override

public **void** run()

{

System.out.println("I can run !!");

}

@Override

public **void** speak()

{

System.out.println("I can speak !!");

}

@Override

public **void** show()

{

System.out.println("I am a person, I can speak and run !!");

}

}

public **class** MultipleInheritance

{

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

{

Person obj = **new** Person();

obj.run();

obj.speak();

obj.show();

}

}

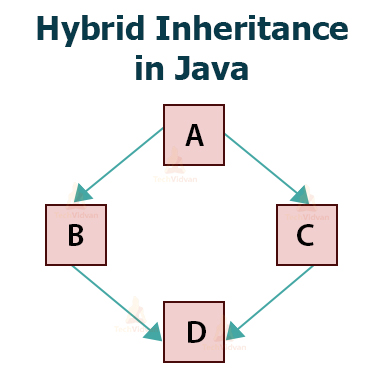
Output:

I can run !!  
I can speak !!  
I am a person, I can speak and run !!

Hybrid inheritance:-

It is a combination of two or more types of inheritance. The hybrid inheritance is also not possible with classes because Java doesn’t support multiple inheritance with classes. We can achieve hybrid inheritance only through Interfaces.

In the following diagram, class A is the base class for subclasses B and C. And, class D inherits both the classes B and C.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/hybrid-inheritance-in-java.jpg)

Example:

package com.techvidvan.inheritance;

//base class 1

**class** Ability

{

public **void** show()

{

System.out.println("I am a person, I can speak and run !!");

}

}

//child interface 1

interface Moveable

{

public **void** run();

}

//child interface2

interface Speakable

{

public **void** speak();

}

//child class inheriting two base interfaces

**class** Person **extends** Ability **implements** Moveable, Speakable

{

@Override

public **void** run()

{

System.out.println("I can run !!");

}

@Override

public **void** speak()

{

System.out.println("I can speak !!");

}

}

public **class** HybridInheritance

{

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

{

Person obj = **new** Person();

obj.run();

obj.speak();

obj.show();

}

}

Output:

I can run !!  
I can speak !!  
I am a person, I can speak and run !!

Preventing inheritance:-

* *You can prevent sub class creation by using final modifier.*
* *If a parent class declared as final we can’t create sub class for that class. final class Parent*

{ }

class Child extends Parent

{ }

***compilation error:- cannot inherit from final Parent***

### Inheritance in OOP with Real-time Example

Consider an application Polygon that represents different types of Shapes.

We are supposed to create two different types of Polygons, one will be Rectangle and the other will be Triangle.

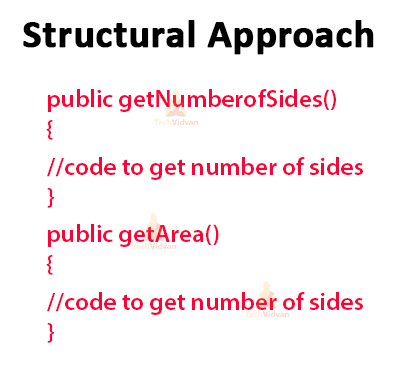
Let’s compare and study the two different approaches of coding with a structured and object-oriented programming perspective.

**Structural approach:**

Using a structured programming approach, we will create two functions:

* One to get the Number of sides of a polygon.
* And the other to calculate the area.

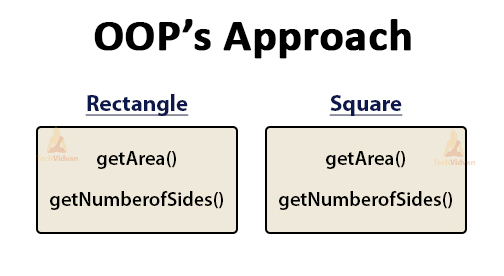
The working of these functions remains the same across two different shapes.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/strucutral-approach.jpg)

**OOP’s approach:**

Using the OOPs programming approach, we would create two different classes.

* Each having implementation of thegetNumberOfSides() and getArea() functions.
* This will reduce extra work.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/OOPs-Approach.jpg)

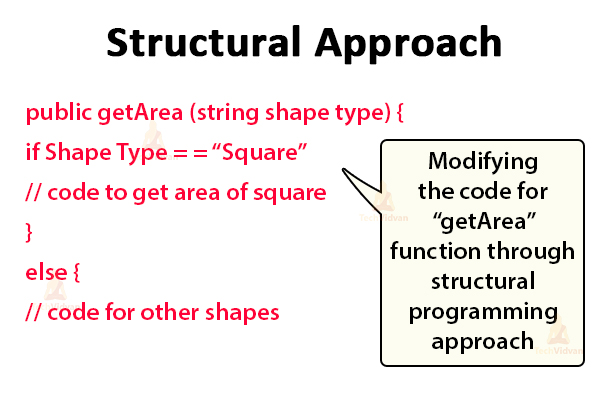
#### Change Request in Software

Suppose there is a change in the functionality of the software. You are supposed to add the functionality of finding the area of a Square.

Let’s see how to deal with this problem with both approaches.

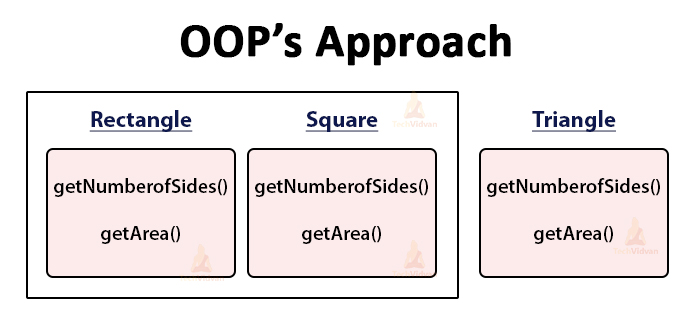
**Structural approach:**

If we want to add a new feature using a functional or traditional methodology, we will need to modify the getArea() function which is already tested and baselined. If we add new functionality of finding the area of a Square then our code will look like:

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/structural-approach-1.jpg)

**OOP’s approach:**

Using the Object-Oriented approach, you just require to add a new class Square which will have the unique functionality of finding the area of Square. There is no need to change the piece of code that is already tested by using this approach.

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/oop-approach.jpg)

#### Another Change Request

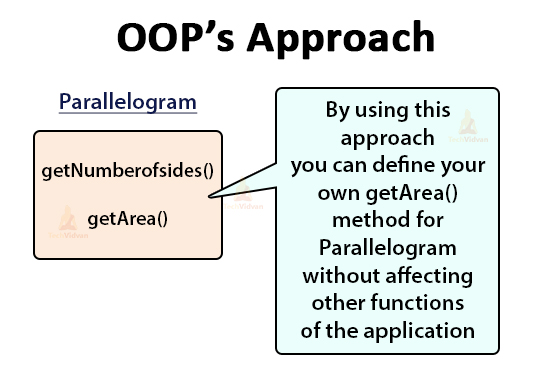
Suppose if there are some more changes required in the software. For example, if you want to add a Shape Parallelogram with its own unique requirements.

**Structural approach:**

If we want to add the Parallelogram Shape in the existing class using the structural approach, then we will definitely need to make changes in the existing code.

**OOP’s approach:**

If we want to add another shape in the existing class using the object-oriented approach, we will just need to create a new class Parallelogram with its unique methods. The diagram below illustrates the same –

[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/OOPs-Approach-1.jpg)

So, even though at the initial stage, the structural programming was appearing to be an easy approach, but as the complexity of the code increases and there are more changes in the requirements then this approach fails.

Eventually, the Object-Oriented approach wins in the long term.

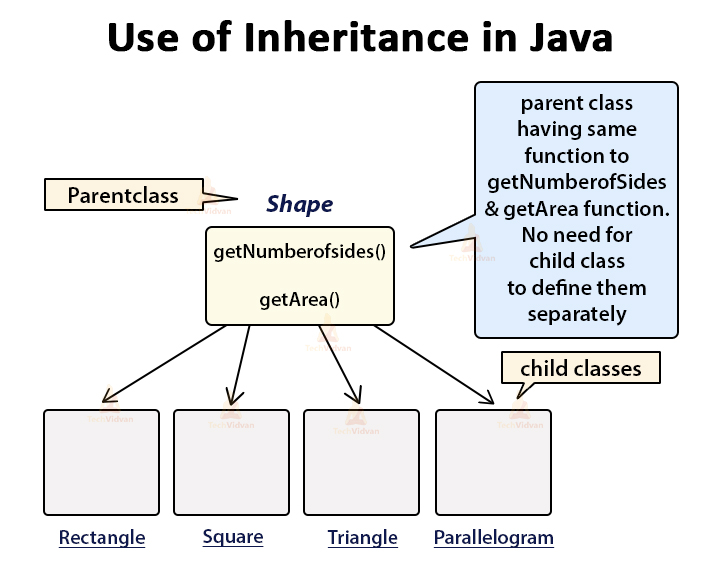
But you might be thinking that across all the classes, we have to write a repeated piece of code for each class.

To overcome this problem of repetition, we can create a parent class called “Shape” and implement the same function of getNumberOfSides and getArea. Then we will create the child classes that will inherit this parent class Shape.

So that they will have access to getNumberOfSides and getArea functions in the Shape class.

There is no need to declare these functions in each class. This concept is called Inheritance in java.



[](https://techvidvan.com/tutorials/wp-content/uploads/sites/2/2020/02/use-of-inheritance-in-java.jpg)

So you can clearly see that with the help of the Inheritance approach of OOP we can easily update our code without disturbing the code which is already tested.

**Polymorphism:**

The word polymorphism is derived from two Greek words: poly and morphs. The word “poly” implies many and “morphs” means forms.

Therefore, polymorphism means “many forms”. That is one thing that can take many forms.

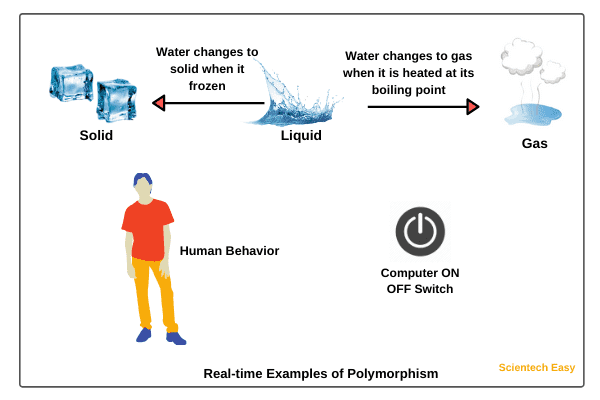
Polymorphism is a concept by which we can perform a single task in different ways. That is, when a single entity behaves differently in different cases, it is called polymorphism in Java.

We can achieve flexibility in our code using polymorphism because we can perform various operations by using methods with the same names according to requirements.

Let’s understand it with some real time examples.

## Real time Example of Polymorphism in Java

There are several real-time examples of polymorphism in the world.

[](https://www.scientecheasy.com/2020/02/polymorphism-in-java.html/)  
1. We all know that water is a liquid, but it changes to solid when it frozen, and it changes to a gas when it is heated at its boiling point.

2. The best example of polymorphism is human behaviour. One person can have different behaviour. For example, a person acts as an employee in the office, a customer in the shopping mall, a passenger in bus/train, a student in school, and a son at home.

3. We all use a single button to switch ON and OFF the computer.

4. A boy starts his love by saying the word “friendship” but the girl ends that love with the same word “friendship”. The girl says that we will be always good friends.

Here, the word “friendship” is the same but attitude is different. This beautiful concept is nothing but polymorphism.

#### ****Polymorphism in Java Example****

A superclass named “Shapes” has a method “area()”. Subclasses of “Shapes” can be “Triangle”, “circle”, “Rectangle”, etc. Each subclass has its way of calculating area. Using Inheritance and Polymorphism means, the subclasses can use the “area()” method to find the area’s formula for that shape.

**Code:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class Shapes {    public void area() {      System.out.println("The formula for area of ");    }  }  class Triangle extends Shapes {    public void area() {      System.out.println("Triangle is ½ \* base \* height ");    }  }  class Circle extends Shapes {    public void area() {      System.out.println("Circle is 3.14 \* radius \* radius ");    }  }  class Main {    public static void main(String[] args) {      Shapes myShape = new Shapes();  // Create a Shapes object      Shapes myTriangle = new Triangle();  // Create a Triangle object      Shapes myCircle = new Circle();  // Create a Circle object      myShape.area();      myTriangle.area();      myShape.area();      myCircle.area();    }  } |

**Output:**

The formula for the area of Triangle is ½ \* base \* height  
The formula for the area of the Circle is 3.14 \* radius \* radius

Example 2:

To take a real time example, we can consider ourself. As a person we have many different forms like student, teacher, player, father/mother etc. The same person can be a teacher as well as a player, so we can say person object is polymorphic in nature.

Another real world example is your mobile. Sometime your mobile behaves as a phone, sometime as a camera, sometime as a radio etc. Here the same mobile phone has different forms, so we can say the mobile object is polymorphic in nature.

## Polymorphism program in Java

class Person {

String name;

void walk() {

System.out.println("My name is "+ name+ " and I can walk");

}

void talk() {

System.out.println("My name is "+ name+ " and I can talk");

}

}

class Teacher extends Person {

String subject;

void teach() {

System.out.println("I teaches "+subject);

}

public static void main(String [] args) {

Teacher teacher = new Teacher(); // teacher object having it's own form

teacher.name = "Rahul";

teacher.subject = "Math";

teacher.walk();

teacher.talk();

teacher.teach();

Person person = teacher; // teacher object takes the form of person object

person.walk();

person.talk();

//person.teach(); // Can't call teach method as it's not in Person class

}

}

**Output:**

My name is Rahul and I can walk

My name is Rahul and I can talk

I teaches Math

My name is Rahul and I can walk

My name is Rahul and I can talk

Here the object of teacher class is polymorphic as it can take two forms, one as itself and other as person. Remember if an object of subclass is behaving as parent class, then it can call the behaviours available in parent class only. In above example once the teacher is behaving as person, it can call only walk and talk behaviour.

## ****Types of Polymorphism****

You can perform Polymorphism in Java via two different methods:

1. Method Overloading (Static polymorphism)
2. Method Overriding (Dynamic polymorphism)

### ****What is Method Overloading in Java?****

**Method overloading** is the process that can create multiple methods of the same name in the same class, and all the methods work in different ways. Method overloading occurs when there is more than one method of the same name in the class.

#### ****Example of Method Overloading in Java****

|  |  |
| --- | --- |
|  | class Shapes {    public void area() {      System.out.println("Find area ");    }  public void area(int r) {      System.out.println("Circle area = "+3.14\*r\*r);    }  public void area(double b, double h) {      System.out.println("Triangle area="+0.5\*b\*h);    }  public void area(int l, int b) {      System.out.println("Rectangle area="+l\*b);    }  }  class Main {    public static void main(String[] args) {      Shapes myShape = new Shapes();  // Create a Shapes object      myShape.area();      myShape.area(5);      myShape.area(6.0,1.2);      myShape.area(6,2);    }  } |

**Output:** Find area  
Circle area = 78.5  
Triangle area=3.60  
Rectangle area=12

### ****What is Method Overriding in Java?****

Method overriding is the process when the subclass or a child class has the same method as declared in the parent class.

### ****Example of Method Overriding in Java****

class Vehicle{

//defining a method

void run(){System.out.println("Vehicle is moving");}

}

//Creating a child class

class Car2 extends Vehicle{

//defining the same method as in the parent class

void run(){System.out.println("car is running safely");}

public static void main(String args[]){

Car2 obj = new Car2();//creating object

obj.run();//calling method

}

}

**Output:**

Car is running safely

Also, Polymorphism in Java can be classified into two types, i.e:

1. Static/Compile-Time Polymorphism
2. Dynamic/Runtime Polymorphism

### ****What is Compile-Time Polymorphism in Java?****

**Compile Time Polymorphism In** Java is also known as **Static Polymorphism.** Furthermore, the call to the method is resolved at compile-time. Compile-Time polymorphism is achieved through **Method Overloading**. This type of polymorphism can also be achieved through **Operator Overloading**. However, Java does not support Operator Overloading.

Method Overloading is when a class has multiple methods with the same name, but the number, types, and order of parameters and the return type of the methods are different. Java allows the user freedom to use the same name for various functions as long as it can distinguish between them by the type and number of parameters.

### ****Example of Compile-Time Polymorphism in Java****

We will do addition in Java and understand the concept of compile time polymorphism using subtract()

package staticPolymorphism;

public class Addition

{

void sum(int a, int b)

{

int c = a+b;

System.out.println(“ Addition of two numbers :” +c); }

void sum(int a, int b, int e)

{

int c = a+b+e;

System.out.println(“ Addition of three numbers :” +c); }

public static void main(String[] args)

{

Addition obj = new Addition();

obj.sum ( 30,90);

obj.sum(45, 80, 22);

}

}

***The output of the program will be:***

Sum of two numbers: 120

Sum of three numbers: 147

In this program, the sum() method overloads with two types via different parameters.

This is the basic concept of compile-time polymorphism in java where we can perform various operations by using multiple methods having the same name.

### ****What is Runtime Polymorphism in Java?****

**Runtime polymorphism** in Java is also popularly known as **Dynamic Binding or Dynamic Method Dispatch.** In this process, the call to an overridden method is resolved dynamically at runtime rather than at compile-time. You can achieve Runtime polymorphism via **Method Overriding**.

Method Overriding is done when a child or a subclass has a method with the same name, parameters, and return type as the parent or the superclass; then that function overrides the function in the superclass. In simpler terms, if the subclass provides its definition to a method already present in the superclass; then that function in the base class is said to be overridden.

Also, it should be noted that runtime polymorphism can only be achieved through functions and not data members.

Overriding is done by using a reference variable of the superclass. The method to be called is determined based on the object which is being referred to by the reference variable. This is also known as **Upcasting**.

Upcasting takes place when the Parent class’s reference variable refers to the object of the child class. For example:

|  |  |
| --- | --- |
| 1  2  3 | class A{}  class B extends A{}  A a=new B(); //upcasting |

#### ****Examples of Runtime Polymorphism in Java****

**Example 1:**

In this example, we are creating one superclass Animal and three subclasses, Herbivores, Carnivores, and Omnivores. Subclasses extend the superclass and override its eat() method. We will call the eat() method by the reference variable of Parent class, i.e. Animal class. As it refers to the base class object and the base class method overrides the superclass method; the base class method is invoked at runtime. As Java Virtual Machine or the JVM and not the compiler determines method invocation, it is, therefore, runtime polymorphism.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class Animal{    void eat(){  System.out.println("Animals Eat");  }  }  class herbivores extends Animal{    void eat(){  System.out.println("Herbivores Eat Plants");  }    }  class omnivores extends Animal{    void eat(){  System.out.println("Omnivores Eat Plants and meat");  }    }  class carnivores extends Animal{    void eat(){  System.out.println("Carnivores Eat meat");  }    }  class main{    public static void main(String args[]){      Animal A = new Animal();      Animal h = new herbivores(); //upcasting      Animal o = new omnivores(); //upcasting      Animal c = new carnivores(); //upcasting      A.eat();      h.eat();      o.eat();      c.eat();    }  } |

**Output:**

Animals eat  
Herbivores Eat Plants  
Omnivores Eat Plants and meat  
Carnivores eat meat

**Example 2:**

In this example, we are creating one superclass Hillstations and three subclasses Manali, Mussoorie, Gulmarg. Subclasses extend the superclass and override its location() and famousfor() method. We will call the location() and famousfor() method by the Parent class’, i.e. Hillstations class. As it refers to the base class object and the base class method overrides the superclass method; the base class method is invoked at runtime. Also, as Java Virtual Machine or the JVM and not the compiler determines method invocation, it is runtime polymorphism.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | class Hillstations{    void location(){  System.out.println("Location is:");  }  void famousfor(){  System.out.println("Famous for:");  }    }  class Manali extends Hillstations {    void location(){  System.out.println("Manali is in Himachal Pradesh");  }  void famousfor(){  System.out.println("It is Famous for Hadimba Temple and adventure sports");  }    }  class Mussoorie extends Hillstations {    void location(){  System.out.println("Mussoorie is in Uttarakhand");  }  void famousfor(){  System.out.println("It is Famous for education institutions");  }    }  class Gulmarg extends Hillstations {    void location(){  System.out.println("Gulmarg is in J&K");  }  void famousfor(){  System.out.println("It is Famous for skiing");  }    }  class main{    public static void main(String args[]){      Hillstations A = new Hillstations();      Hillstations M = new Manali();   Hillstations Mu = new Mussoorie();   Hillstations G = new Gulmarg();   A.location();  A.famousfor();  M.location();  M.famousfor();  Mu.location();  Mu.famousfor();    G.location();  G.famousfor();    }  } |

**Output:**

Location is:  
Famous for:  
Manali is in Himachal Pradesh  
It is Famous for Hadimba Temple and adventure sports  
Mussoorie is in Uttarakhand  
It is Famous for education institutions  
Gulmarg is in J&K  
It is Famous for skiing

## Example 3:

Let’s take an example program where we will implement static polymorphism. In this example program, we will create a class StaticPolymorphism.

In this class, we will create two methods having the same name sum. Both of these methods will have different signatures. Let’s start coding.

**Program source code 1:**

package staticPolymorphism;

public class StaticPoly

{

void sum(int x, int y)

{

int s = x + y;

System.out.println("Sum of two numbers: " +s);

}

void sum(int x, int y, int z)

{

int s = x + y + z;

System.out.println("Sum of three numbers: " +s);

}

public static void main(String[] args)

{

StaticPoly obj = new StaticPoly();

obj.sum(20, 10);

obj.sum(10, 20, 30);

}

}

Output:

Sum of two numbers: 30

Sum of three numbers: 60

Sum of three numbers: 60

**Explanation:**

As you can observe in the preceding example program, the sum() method is overloaded two times because both methods’ signatures vary in the number of parameters.

The first sum() method accepts two parameters whereas, the second sum() method accepts three parameters.

When we will call the first sum() method using reference variable “obj” by passing two int type argument values, Java compiler binds the definition of sum(int x, int y) method with sum(20, 10) method during compilation and calls the appropriate method.

Hence, the sum of two numbers is displayed by invoking sum() method on the console.

Similarly, when we call the second sum() method by passing three int type argument values, Java compiler binds the definition of sum(int x, int y, int z) with sum(10, 20, 30) method during compilation, and calls a method of the sum of three numbers.

Thus, Java compiler matches the values passed to a method during compilation, binds method call with appropriate method definition, and calls the appropriate method.

In this way, compile-time polymorphism allows us to perform various operations by using multiple methods with the same name.

**Key point:**

Java compiler differentiates multiple methods having the same name by their signatures

### ****Example of run-time polymorphism in java****

We will create two classes Car and Innova, Innova class will extend the car class and will override its run() method.

class Car

{

void run()

{

System.out.println(“ running”);

}

}

class innova extends Car

{

void run();

{

System.out.println(“ running fast at 120km”);

}

public static void main(String args[])

{

Car c = new innova();

c.run();

}

}

The output of the following program will be;

Running fast at 120 km.

Another example for run-time polymorphism in Java

Now, let us check if we can achieve runtime polymorphism via data members.

class car

{

int speedlimit = 125;

}

class innova extends car

{

int speedlimit = 135;

public static void main(String args[])

{

car obj = new innova();

System.out.println(obj.speedlimit);

}

***The output of the following program will be :***

125

This clearly implies we can’t achieve Runtime polymorphism via data members. In short, a method is overridden, not the data members.

### ****Runtime polymorphism with multilevel inheritance****

class grandfather

{

void swim()

{

System.out.println(“ Swimming”);

}

}

class father extends grandfather

{

void swim()

{

System.out.println(“ Swimming in river”);

}

}

class son extends father

{

void swim()

{

System.out.println(“ Swimming in pool”);

}

public static void main(String args[])

{

grandfather f1,f2,f3;

f1 =new grandfather();

f2 = new father();

f3 = new son();

f1.swim();

f2.swim();

f3.swim():

}

}

***The output of the following program will be:***

Swimming, Swimming in river, Swimming in pool

**Another runtime polymorphism with multilevel inheritance example**

class soundAnimal

{

public void Sound()

{

System.out.println("Different sounds of animal"); }

}

class buffalo extends soundAnimal

{

public void Sound()

{

System.out.println("The buffalo sound- gho,gho"); }

}

class snake extends soundAnimal

{

public void Sound()

{

System.out.println("The snake sound- his,his"); }

}

class tiger extends soundAnimal

{

public void Sound()

{

System.out.println("The tiger sounds- roooo, rooo"); }

}

public class Animal Main

{

public static void main(String[] args)

{

soundAnimal Animal = new soundAnimal(); soundAnimal buffalo = new buffalo();

soundAnimal snake = new snake();

soundAnimal tiger = new tiger();

Animal.Sound();

buffalo.Sound();

snake.Sound();

tiger.Sound();

}

}

***The output of the following program will be;***

The buffalo sound- gho,gho

The snake sound- his,his

The tiger sound- roooo,roooo

**Abstraction:**

**Abstraction** is the concept of exposing only the required essential characteristics and behavior with respect to a context.

Hiding of data is known as **data abstraction**. In object oriented programming language this is implemented automatically while writing the code in the form of class and object.

### Real Life Example of Abstraction in Java

Abstraction shows only important things to the user and hides the internal details, for example, when we ride a bike, we only know about how to ride bikes but can not know about how it work? And also we do not know the internal functionality of a bike.



Another real life example of Abstraction is ATM Machine; All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement…etc. but we can't know internal details about ATM.



**Note:**Data abstraction can be used to provide security for the data from the unauthorized methods.

**Note:**In Java language data abstraction can achieve using class.

## Example of Abstraction

**class** Customer

{

**int** account\_no;

**float** balance\_Amt;

String name;

**int** age;

String address;

**void** balance\_inquiry()

{

/\* to perform balance inquiry only account number

is required that means remaining properties

are hidden for balance inquiry method \*/

}

**void** fund\_Transfer()

{

/\* To transfer the fund account number and

balance is required and remaining properties

are hidden for fund transfer method \*/

}

There are two types of methods in java

* 1. **Normal methods**
  2. **Abstract methods**

***Normal methods:- (component method/concrete method)***

Normal method is a method which contains method declaration as well as method implementation.

***Example:-***

*void m1()* ***--->method declaration***

*{ body;* ***--->method implementation***

}

***Abstract methods:-***

1. The method which is having only method declaration but not method implementations such type of methods are called abstract Methods.
2. In java every abstract method must end with “ ; ”.

***Example : - abstract void m1 ();***  ***method declaration***

**Based on above representation of methods the classes are divided into two types**

1. **Normal classes.**
2. **Abstract classes.**

***Normal classes:-***

Normal class is a ordinary java class it contains only normal methods if we are trying to declare at least one abstract method that class will become abstract class.

***Example:-***

*class Test* ***//normal class***

*{ void m1() { body ; }* ***//normal method*** *void m2() { body ; }* ***//normal method*** *void m3() { body ; }* ***//normal method***

*};*

***Abstract class:-***

Abstract class is a java class which contains at least one abstract method(wrong definition).

If any abstract method inside the class that class must be abstract.

**Example 1:-**

*class Test* ***//abstract class***

{

*void m1 ( ) { }* ***//normal method*** *void m2( ) { } //****normal method*** *void m3();* ***//abstract method***

};

***Example-2:-***

*class Test* ***//abstract class***

{

*abstract void m1();* ***//abstract method*** *abstract void m2();* ***//abstract method*** *abstract void m3();* ***//abstract method***

};

***Abstract modifier:-***

* Abstract modifier is applicable for methods and classes but not for variables.
* To represent particular class is abstract class and particular method is abstract method to the compiler use abstract modifier.
* The abstract class contains declaration of methods it says abstract class partially implement class hence for partially implemented classes object creation is not possible. If we are trying to

create object of abstract class compiler generate error message “class is abstract con not be instantiated”

***Example -1:-***

* ***Abstract classes are partially implemented classes hence object creation is not possible.***
* ***For the abstract classes object creation not possible, if you are trying to create object compiler will generate error message.***

*abstract class Test //abstract class*

*{ abstract void m1(); //abstract method abstract void m2(); //abstract method abstract void m3(); //abstract method*

*void m4(){System.out.println("m4 method");} //normal method*

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

*{ Test t = new Test(); t.m4();*

*}*

*};*

*Compilation error:- Test is abstract; cannot be instantiated Test t = new Test();*

***Example-2 :*-**

* + Abstract class contains abstract methods for that abstract methods provide the implementation in child classes.
  + **Provide the implementations is nothing but override the methods in child classes.**
  + The abstract class contains declarations but for that declarations implementation is present in child classes.

*abstract class Test*

*{ abstract void m1(); abstract void m2(); abstract void m3();*

*void m4(){System.out.println("m4 method");}*

*};*

*class Test1 extends Test*

*{ void m1(){System.out.println("m1 method");} void m2(){System.out.println("m2 method");} void m3(){System.out.println("m3 method");}*

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

*{ Test1 t = new Test1();*

*t.m1(); t.m2(); t.m3(); t.m4();*

*Test t1 = new Test1(); //abstract class reference variable Child class object*

*t1.m1(); //compile : Test runtime : Test1 t1.m2(); //compile : Test runtime : Test1 t1.m3() ; //compile : Test runtime : Test1 t1.m4() ; //compile : Test runtime : Test1*

*}*

*};*

***Example -3 :-***

* + Abstract class contains abstract methods for that abstract methods provide the implementation in child classes.
  + **if the child class is unable to provide the implementation of all parent class abstract methods at that situation declare that class with abstract modifier then take one more child class to complete the implementation of remaining abstract methods.**
  + It is possible to declare multiple child classes but at final complete the implementation of all methods.

abstract class Test

{ abstract void m1(); abstract void m2(); abstract void m3();

void m4(){System.out.println("m4 method");}

};

abstract class Test1 extends Test

{ void m1(){System.out.println("m1 method");}

};

abstract class Test2 extends Test1

{ void m2(){System.out.println("m2 method");}

};

class Test3 extends Test2

{ void m3(){System.out.println("m3 method");} public static void main(String[] args)

{ Test3 t = new Test3(); t.m1();

t.m2();

t.m3();

t.m4();

}

};

**Example :-** inside the abstract class it is possible to declare abstract class Test

*{ public int a=10; public final int b=20;*

*public static final int c=30; void disp1()*

*{ System.out.println("a value is="+a);*

*}*

*};*

*class Test1 extends Test*

*{ void disp2()*

*{ System.out.println("b value is="+b); System.out.println("c value is="+c);*

*}*

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

*{ Test1 t = new Test1();*

t.disp1();

t.disp2();

}

};

***Example-5 :-***

***for the abstract methods it is possible to provide any return type(void, int, char,Boolean…..etc)***

*class Emp{}; abstract class Test1*

*{ abstract int m1(char ch); abstract boolean m2(int a); abstract Emp m3();*

*}*

*abstract class Test2 extends Test1*

*{ int m1(char ch)*

*{ System.out.println("char value is:-"+ch); return 100;*

*}*

*};*

*class Test3 extends Test2*

*{ boolean m2(int a)*

*{ System.out.println("int value is:-"+a); return true;*

*}*

*Emp m3()*

*{ System.out.println("m3 method"); return new Emp();*

*}*

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

*{ Test3 t=new Test3(); int a=t.m1('a');*

*System.out.println("m1() return value is:-"+a); boolean b=t.m2(111); System.out.println("m2() return value is:-"+b); Emp e = t.m3();*

*System.out.println("m3() return value is:-"+e);*

*}*

*};*

***Example-6:-***

***It is possible to override non-abstract as a abstract method in child class.***

abstract class Test

*{ abstract void m1();* ***//m1() abstract method***

*void m2(){System.out.println("m2 method");}* ***//m2() normal method***

};

abstract class Test1 extends Test

*{ void m1(){System.out.println("m1 method");}* ***//m1() normal method abstract void m2(); //m2() abstract method***

};

class FinalClass extends Test1

{ void m2(){System.out.println("FinalClass m2() method");} public static void main(String[] args)

{ FinalClass f = new FinalClass(); f.m1();

f.m2();

}

};

***Example:-***

abstract class Test

{ public static void main(String[] args)

{ System.out.println("this is abstract class main");

}

};

***Example-8:-***

* + *Constructor is used to create object (wrong definition).*
  + Constructor is executed during object creation to initialize values to instance variables.
  + Constructors are used to write the write the functionality that functionality executed during object creation.
  + There are multiple ways to crate object in java but if we are crating object by using “new” then only constructor executed.

***Note :- in below example abstract class constructor is executed but object is not created.***

abstract class Test

{ Test()

{ System.out.println("abstrac calss con");

}

};

class Test1 extends Test

{ Test1()

{ super();

System.out.println("normal class con");

}

public static void main(String[] args)

{ new Test1();

}

};

D:\>java Test1 abstrac calss con normal class con

case 1:- [abstract method to normal method] abstract class Test

{ abstract void m1();

};

class Test1 extends Test

{ void m1(){System.out.println("m1 method");}

};

case 2:-[normal method to abstract method] class Test

{ void m1(){System.out.println("m1 method");}

};

abstract class Test1 extends Test

{ abstract void m1();

};

***Example-9:- the abstract class allows zero number of abstract method. Definition of abstract class:-***

***Abstract class may contains abstract methods or may not contains abstract methods but object creation is not possible. The below example contains zero number of abstract methods.***

***Ex:- HttpServlet (doesn’t contains abstract methods still it is abstract object creation not possible )***

*abstract class Test*

*{ void cm() { System.out.println("ratan"); }*

*void pm() { System.out.println("anushka"); }*

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

*{ Test t = new Test(); t.cm(); t.pm();*

*}*

*};*

*Test.java:6: Test is abstract; cannot be instantiated*

***Abstraction definition :-***

* + The process highlighting the set of services and hiding the internal implementation is called abstraction.
  + Bank ATM Screens Hiding the internal implementation and highlighting set of services like , money transfer, mobile registration,…etc).
  + Syllabus copy of institute just highlighting the contents of java but implementation there in classed rooms .
  + We are achieving abstraction concept by using Abstract classes & Interfaces.

### Java Program of abstraction using Interface

The classes in below program implements the Shape interface that we discussed above.

class Circle implements Shape {

private double radius;

public Circle(double r) {

this.radius = r;

}

public void draw() {

System.out.println("Drawing Circle");

}

public double getArea() {

return Math.PI\*radius\*radius;

}

}

class Rectangle implements Shape {

private double length;

private double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

public void draw() {

System.out.println("Drawing Rectangle");

}

public double getArea() {

return length\*width;

}

}

Similar to *Circle* and *Rectangle* there can be other shape type as well that can implement *Shape* interface. Now in order to access the features(*getArea, draw*) of a shape, all you need to know is the implementing class, you don't need to know how they have implemented the features, that can be hidden from you, this is what abstraction is. The class given below accesses these features.

public class AbstractionTest {

public static void main (String args[]) {

// Shape object referring to circle.

Shape circle = new Circle(10);

circle.draw();

System.out.println("Area of given circle = "+circle.getArea());

// Shape object referring to rectangle.

Shape rect = new Rectangle(10,10);

rect.draw();

System.out.println("Area of given rectangle = "+rect.getArea());

}

}

**Output:**

Drawing Circle

Area of given circle = 314.1592653589793

Drawing Rectangle

Area of given rectangle = 100.0

In order to run above program successfully, ensure that you have declared all the classes and interfaces given above inside same directory.

### Java Program of abstraction using abstract class

The program below contains an abstract class TwoDShape which contains one abstract and one concrete method.

abstract class TwoDShape {

private double length;

private double width;

public TwoDShape(double length, double width) {

this.length = length;

this.width = width;

}

// Declaring abstract method

public abstract void draw();

// Defining concrete method

public double getArea() {

return length\*width;

}

}

public class Rectangle extends TwoDShape {

public Rectangle(double length, double width) {

super(length,width);

}

public void draw() {

System.out.println("Drawing Rectangle");

}

public static void main (String args[]) {

// TwoDShape object referring to rectangle.

TwoDShape rect = new Rectangle(10,10);

rect.draw();

System.out.println("Area of given rectangle = "+rect.getArea());

}

}

**Output:**

Drawing Rectangle

Area of given rectangle = 100.0

Here the abstract class TwoDShape gives the definition of getArea method while hides the implementation of draw method. The implementation of draw method is given by the implementing class which is Rectangle class. So for us the TwoDShape class is an example of partial abstraction.

### Advantages of Data Abstraction in Java

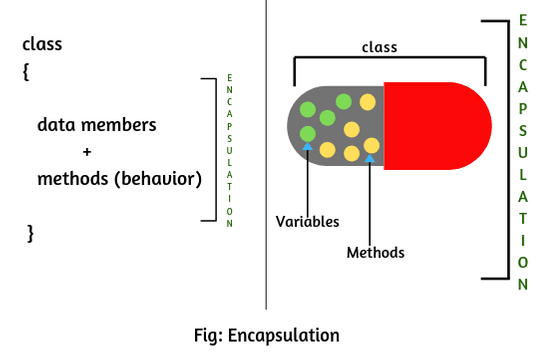
Here we discuss some of its benefits in order to acknowledge its significance. The benefits of data abstraction are given below;

* Data abstraction increases the reusability of the code by avoiding any chances of redundancy.
* It increases the readability of the code as it eliminates the possibility of displaying the complex working of the code.
* With the implementation of classes and objects, comes enhanced security. Since data abstraction is a method of implementing classes and objects any denying access to other classes of accessing the data members and member functions of the base class.
* It helps the user to write a high-level code.
* It separates the entire program into code and implementation making it more comprehensible.
* Helps the user to avoid writing the low level code.
* Avoids code duplication and increases reusability.
* Can change internal implementation of class independently without affecting the user.
* Helps to increase security of an application or program as only important details are provided to the user.

**Encapsulation:**

The process of binding the data and code as a single unit is called encapsulation.

We are able to provide more encapsulation by taking the private data(variables) members. To get and set the values from private members use getters and setters to set the data and to get the data.



**Example:-**

class Encapsulation

{ private int sid; private int sname;

//mutator methods

*public void setSid(int x)*

*{ this.sid=sid; }*

*public void setSname(String sname)*

*{ this.sname=sname; }*

//Accessor methods

*public int getSid()*

*{ return sid; } public String getSname()*

*{ return sname; }*

*};*

To access encapsulated use fallowing code:-

*class Test*

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

*{ Encapsulation e=new Encapsulation(); e.setSid(100);*

*e.setSname("ratan"); System.out.println(e.getSid()); System.out.println(e.getSname());*

*}*

*};*

**Realtime Example 1:**  
School bag is one of the most real examples of Encapsulation. School bag can keep our books, pens, etc.

**Realtime Example 2:**  
When you log into your email accounts such as Gmail, Yahoo Mail, or Rediff mail, there is a lot of internal processes taking place in the backend and you have no control over it.

When you enter the password for logging, they are retrieved in an encrypted form and verified, and then you are given access to your account.

You do not have control over it that how the password has been verified. Thus, it keeps our account safe from being misused.

**Realtime Example 3:**  
Suppose you have an account in the bank. If your balance variable is declared as a public variable in the bank software, your account balance will be known as public, In this case, anyone can know your account balance. So, would you like it? Obviously No.

So, they declare balance variable as private for making your account safe, so that anyone cannot see your account balance.

The person who has to see his account balance, will have to access only private members through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.

Thus, we can achieve security by utilizing the concept of data hiding. This is called Encapsulation in Java.

## How to achieve or implement Encapsulation in Java:

There are two important points whereby we can achieve or implement encapsulation in Java program.

1. Declaring the instance variable of the class as private. so that it cannot be accessed directly by anyone from outside the class.

2. Provide the public setter and getter methods in the class to set/modify the values of the variable/fields.

## Advantage of Encapsulation in Java:

There are following advantages of encapsulation in Java. They are as follows:

1. The encapsulated code is more flexible and easy to change with new requirements.  
2. It prevents the other classes to access the private fields.  
3. Encapsulation allows modifying implemented code without breaking other code that has implemented the code.  
4. It keeps the data and codes safe from external inheritance. Thus, Encapsulation helps to achieve security.  
5. It improves the maintainability of the application.  
6. If you don’t define the setter method in the class then the fields can be made read-only.  
7. If you don’t define the getter method in the class then the fields can be made write-only.

## Disadvantage of Encapsulation in Java

The main disadvantage of encapsulation in Java is it increases the length of the code and slows shutdown execution.

## Data Hiding in Java:

**Data hiding in Java** is an important principle of [object-oriented programming system (OOPs)](https://www.scientecheasy.com/2020/07/oops-concepts-in-java.html/). It prevents to access data members (variables) directly from outside the class so that we can achieve security on data. This oops feature is called data hiding in Java.  
  
An outside person could not access our internal data directly or our internal data should not go out directly. After validation or authentication, the outside person can access our internal data.

For example, after providing proper username and password, you can able to access your Gmail inbox information.

## How to achieve Data hiding programmatically?

By declaring data members (variables) as private, we can achieve or implement data hiding. If the variables are declared as private in the class, nobody can access them from outside the class.

The biggest advantage of data hiding is we can achieve security.

**Key points:**

1. It is highly recommended to declare data members as private in the class.

2. A combination of data hiding and abstraction is nothing but encapsulation.

**Encapsulation** = **Data Hiding** + [**Abstraction**](https://www.scientecheasy.com/2020/05/java-abstraction.html/)

If any component follows data hiding and abstraction, it is called an encapsulated component.

## Tightly Encapsulated Class in Java

If each variable is declared as private in the class, it is called **tightly encapsulated class in Java**. For tightly encapsulated class, we are not required to check whether class contains getter and setter method or not and whether these methods are declared as public or not.

For example:

public class Account

{

private double balance;

public double getbalance()

 {

   return balance;

 }

}

## Example:

package com.dataflair.encapsulation;

**class** DataFlair {

private String course;

public String getCourse() {

**return** course;

}

public **void** setCourse(String s) {

**this**.course = s;

}

}

public **class** Encapsulation {

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

DataFlair df = **new** DataFlair();

df.setCourse("Java");

System.out.println(df.getCourse());

//System.out.println(DataFlair.course);

}

}

**Output:**

Java

Now before you proceed further there are a few things that you have to notice in this program.

a. We have commented out one line. That line tries to access the private variable outside the main class. Hence if you uncomment that you will see that the compiler returns an error.

b. The access of the methods is intentionally set to “public” for the class Encapsulation to be able to access it. Change the access specifier to private and you will see that the compiler returns an error of inaccessibility.

c. This essentially means that you can change the access specifier to private if you want the method calls to be from inside the class only.

### Ways to achieve Encapsulation in Java

There are a few ways by which we can achieve encapsulation in java. Some of them are:

a. Declaring the class variables as private so that they are inaccessible from outside the scope of the class.  
b. Designing getter and setter methods for the class and using them accordingly. The presence of getter and setter methods in a class define what type of class it is.

**Java program to illustrate the implementation of encapsulation by using private variables:**

package com.dataflair.encapsulation;

public **class** TestEncapsulation {

private String privateVar;

TestEncapsulation() {

privateVar = "java";

}

}

public **class** PrivateVariables {

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

System.out.println(TestEncapsulation.privateVar);

}

}

**Output:**

Privatevariables.java:12: error: privateVar has private access in TestEncapsulation  
System.out.println(TestEncapsulation.privateVar);

As you can see the class cannot access the variable from the previous class. This is an example of the private access specifier.

**Java program to illustrate the use of getter and setter methods in Java:**

package com.dataflair.encapsulation;

**class** TestEncapsulation {

private String privateVar;

TestEncapsulation() {

privateVar = "java";

}

public **void** getVariable() {

System.out.println(privateVar);

}

public **void** setVariable(String setvalue) {

privateVar = setvalue;

}

}

public **class** PrivateVariables {

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

TestEncapsulation test = **new** TestEncapsulation();

test.setVariable("Python");

test.getVariable();

//System.out.println(TestEncapsulation.privateVar);

}

}

Output

Python

**Real-life example of Java Encapsulation:**

package com.dataflair.encapsulation

**class** Remote {

private boolean OnValue;

Remote() {

OnValue = **false**;

}

public **void** changeOnValue() {

System.out.println("The OnValue of the remote has toggled. Check TV ");

OnValue = !OnValue;

}

}

public **class** TvRemote {

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

Remote remote = **new** Remote();

remote.changeOnValue(); //Equivalent to pressing the "Switch On" button on the remote once;

remote.changeOnValue();

remote.changeOnValue();

}

}

Output

The OnValue of the remote has toggled. Check TV  
The OnValue of the remote has toggled. Check TV  
The OnValue of the remote has toggled. Check TV

In this example, even though you can see that the changeOnValue is changing the value of the OnValue of the remote. But you can’t see its actual value. You can only see if the TV is on or off. This is because there are no getter functions for the remote class. Hence this class is a “Write-Only” class. This is a very important point to consider while understanding encapsulation.

## Difference between Encapsulation and Abstraction in Java:

Abstraction deals with hiding the details and showing the essential things to the user whereas encapsulation binds your data and code together as a single unit.

Encapsulation is not providing full security because we can access private member of the class using reflection API, but in case of Abstraction we can't access static, abstract data member of a class.

In java you can say it takes all your methods, variables and bind them together in a single class.

Abstraction is implemented in Java using interface and abstract class while Encapsulation is implemented using private, package-private and protected access modifier.