**OOP Concept in Java**

OOP stands for Object Oriented Programming Language. OOP concept is one of most important concepts in programming languages. OOPs concept has four major principles which are Abstraction, Inheritance, Encapsulation and Polymorphism. Using this principles user can interact with real world entities. OOPs concept allows users/programmers to create objects and create methods to perform actions on that particular object. The basic concept of OOP is to create objects and re-use them throughout the program and manipulate this object to get result.

**List of OOPs Concepts in Java with Examples:**

**Class**

Class is a group of similar entities. Which contains variables and methods. A class is a user defined blueprint or prototype from which objects are created.  It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order

**Modifiers**: A class can be public or has default access

**class keyword:**class keyword is used to create a class.

**Class name:** The name should begin with an initial letter (capitalized by convention).

**Superclass (if any):** The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

**Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

**Body:** The class body surrounded by braces, {}.

Constructors are used for initializing new objects.

Syntax: Class\_name Object = new Class\_Constructor ();

Fields are variables that provides the state of the class and its objects, and methods are used to implement the behaviour of the class and its objects. There are various types of classes that are used in real time applications such as [nested classes](https://www.geeksforgeeks.org/inner-class-java/), [anonymous classes](https://www.geeksforgeeks.org/anonymous-inner-class-java/), [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/).

**Object**

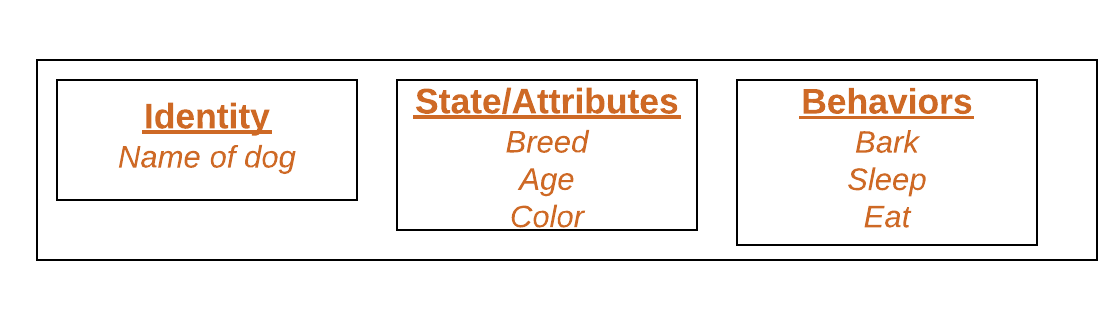
An object can be defined as an instance of a class, and there can be multiple instances of a class in a program. An Object is one of the Java OOPs concepts which contains both the data and the function, which operates on the data. For example - chair, bike, marker, pen, table, car, etc.

An object consists of:

**State**: It is represented by attributes of an object. It also reflects the properties of an object.

**Behaviour**: It is represented by methods of an object. It also reflects the response of an object with other objects.

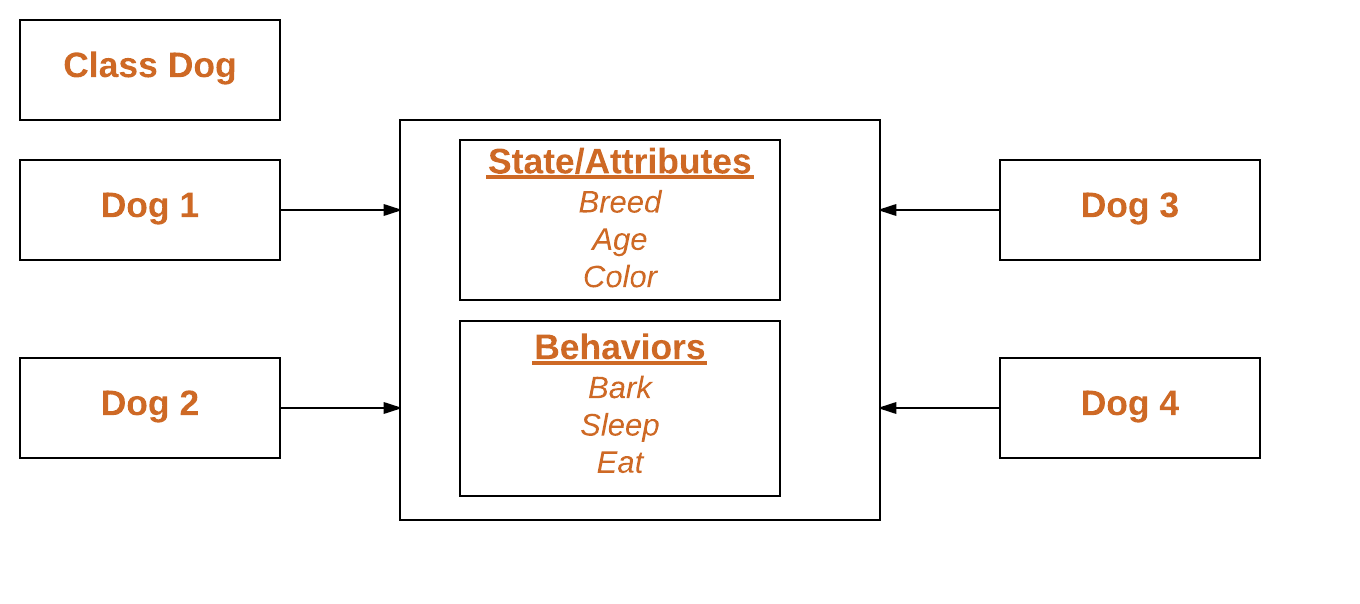
**Identity**: It gives a unique name to an object and enables one object to interact with other objects.



Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Declaring objects (Also called Instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behaviour of the class. But the values of those attributes, i.e., the state are unique for each object. A single class may have any number of instances.



As we declare variables like (type name;). This notifies the compiler that we will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So, for reference variable, type must be strictly a concrete class name. In general, we **can’t** create objects of an abstract class or an interface.

**Initializing an Object:** The **new** operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

package OOPs;

public class Animal {

// global variables

String name;

public static void main(String[] args) {

Animal dog = new Animal();

dog.Sound();

}

public void Sound() {

System.***out***.println("bho..bhao");

}

}

Output: bho..bhao

**Inheritance**

Inheritance is one of the Basic Concepts of OOPs in which one object acquires the properties and behaviours of the parent object. It’s creating a parent-child relationship between two classes. It offers robust and natural mechanism for organizing and structure of any software.

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) 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.

**Why use Inheritance in Java**

For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (*so*[*runtime polymorphism*](https://www.javatpoint.com/runtime-polymorphism-in-java)*can be achieved. Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So, polymorphism means many forms. There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding*.).

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

In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

For Code Reusability.

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.

**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.



**Note: Multiple Inheritance is not supported in Java through class**

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



**Single Inheritance Example**

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

*File: TestInheritance.java*

package OOPs;

class Animal{

public static void main(String args[]){

Dog d=new Dog();

d.bark();

d.eat();

}

void eat(){

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

}

}

class Dog extends Animal{

void bark(){

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

}

}

Output:

barking...

eating...

**Multilevel Inheritance Example**

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

*File: TestInheritance2.java*

class Animal{

public static void main(String args[]){

BabyDog d=new BabyDog();

d.weep();

d.bark();

d.eat();

}

void eat(){

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

}

}

class Dog extends Animal{

void bark(){

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

}

}

class BabyDog extends Dog{

void weep(){

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

}

}

Output:

weeping...

barking...

eating...

**Hierarchical Inheritance Example**

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

*File: TestInheritance3.java*

**class** Animal{

**void** eat(){System.out.println("eating...");}

}

**class** Dog **extends** Animal{

**void** bark(){System.out.println("barking...");}

}

**class** Cat **extends** Animal{

**void** meow(){System.out.println("meowing...");}

}

**class** TestInheritance3{

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

Cat c=**new** Cat();

c.meow();

c.eat();

//c.bark();//C.T.Error

}}

Output:

meowing...

eating...

**Q). Why Multiple Inheritance in not supported in Java.?**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So, whether you have same method or different, there will be compile time error.

**class** A{

**void** msg(){

System.out.println("Hello");}

}

**class** B{

**void** msg(){

System.out.println("Welcome");}

}

**class** C **extends** A, B {

//suppose if it were

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

   C obj=**new** C();

   obj.msg();//Now which msg() method would be invoked?

}

}

Output:

Compile Time Error

**Polymorphism**

Polymorphism refers to one of the OOPs concepts in Java which is the ability of a variable, object or function to take on multiple forms. For example, in English, the verb *run* has a different meaning if you use it with *a laptop*, *a foot race*, and *business*. Here, we understand the meaning of *run* based on the other words used along with it. The same also applied to Polymorphism.

In other word, Polymorphism is used to perform single action in different ways.

let’s say we have a class Animal that has a method sound (). Since this is a generic class so we can’t give it an implementation like: Roar, Meow, Oink etc. We had to give a generic message.

public class Animal{

...

public void sound(){

System.out.println("Animal is making a sound");

}

}

Now let’s say we two subclasses of Animal class: Horse and Cat that extends (see Inheritance) Animal class. We can provide the implementation to the same method like this:

public class Horse extends Animal{

...

@Override

public void sound() {

System.out.println("Neigh");

}

}

**and**

**and**

public class Cat extends Animal {

...

@Override

public void sound() {

System.out.println("Meow");

}

}

As you can see that although we had the common action for all subclasses sound() but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways). It would not make any sense to just call the generic sound() method as each Animal has a different sound. Thus, we can say that the action this method performs is based on the type of object.

**What is Polymorphism Programming.?**

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations. As we have seen in the above example that we have defined the method sound() and have the multiple implementations of it in the different-2 sub classes.  
Which sound() method will be called is determined at runtime so the example we gave above is a **runtime polymorphism example**.

Types of polymorphism and method overloading & overriding are covered in the separate tutorials. You can refer them here:  
1. [Method Overloading in Java](https://beginnersbook.com/2013/05/method-overloading/) – This is an example of compile time (or static polymorphism)  
2. [Method Overriding in Java](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) – This is an example of runtime time (or dynamic polymorphism)  
3. [Types of Polymorphism – Runtime and compile time](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/)

**Example 1: Polymorphism in Java**

**Runtime Polymorphism (**Method Overriding**) it is also known as Dynamic polymorphism.**

Animal.java

public class Animal{

public void sound(){

System.out.println("Animal is making a sound");

}

}

Horse.java

class Horse extends Animal{

@Override

public void sound(){

System.out.println("Neigh");

}

public static void main(String args[]){

Animal obj = new Horse();

obj.sound();

}

}

Output:

Neigh

Cat.java

public class Cat extends Animal{

@Override

public void sound(){

System.out.println("Meow");

}

public static void main(String args[]){

Animal obj = new Cat();

obj.sound();

}

}

Output:

Meow

**Example 2: Polymorphism in Java**

**Compile time polymorphism (**Method Overloading**) It is also known as static polymorphism**

class Overload

{

void demo (int a)

{

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

}

void demo (int a, int b)

{

System.out.println ("a and b: " + a + "," + b);

}

double demo(double a) {

System.out.println("double a: " + a);

return a\*a;

}

}

class MethodOverloading

{

public static void main (String args [])

{

Overload Obj = new Overload();

double result;

Obj .demo(10);

Obj .demo(10, 20);

result = Obj .demo(5.5);

System.out.println("O/P : " + result);

}

}

Here the method demo() is overloaded 3 times: first method has 1 int parameter, second method has 2 int parameters and third one is having double parameter. Which method is to be called is determined by the arguments we pass while calling methods. This happens at compile time so this type of polymorphism is known as compile time polymorphism.

**Output:**

a: 10

a and b: 10,20

double a: 5.5

O/P: 30.25

**Abstraction**

Abstraction is one of the OOP Concepts in Java which is an act of representing essential features without including background details. It is a technique of creating a new data type that is suited for a specific application. Let’s understand this one of the OOPs Concepts with example, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.

**Abstract class in Java with example**

A class that is declared using “**abstract**” keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (regular methods with body). A normal class (non-abstract class) cannot have abstract methods.

An abstract class cannot be **instantiated**, which means you are not allowed to create an **object** of it. Why? We will discuss that later in this guide.

**Why we need an abstract class.?**

Let’s say we have a class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like Lion class will say “Roar” in this method and Dog class will say “Woof”.

So, when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract, we force all the sub classes to implement this method (otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensure that every animal has a sound.

**Abstract Class Example**

//abstract parent class

abstract class Animal{

//abstract method

public abstract void sound();

}

//Dog class extends Animal class

public class Dog extends Animal{

public void sound(){

System.out.println("Woof");

}

public static void main(String args[]){

Animal obj = new Dog();

obj.sound();

}

}

Output:

Woof

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

**Abstract Class Declaration**

An abstract class outlines the methods but not necessarily implements all the methods.

//Declaration using abstract keyword

abstract class A{

//This is abstract method

abstract void myMethod();

//This is concrete method with body

void anotherMethod(){

//Does something

}

}

**Rules**

**Note 1:** As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

**Note 2:** Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods (those that were abstract in parent but implemented in child class).

**Note 3:** If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

**Do you know?** Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

[Interfaces](https://beginnersbook.com/2013/05/java-interface/) on the other hand are used for 100% abstraction (See more about [abstraction](https://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/) here).

**Why we can’t create an object of abstract class.?**

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen? There would be no actual implementation of the method to invoke.  
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

abstract class AbstractDemo{

public void myMethod(){

System.out.println("Hello");

}

abstract public void anotherMethod();

}

public class Demo extends AbstractDemo{

public void anotherMethod() {

System.out.print("Abstract method");

}

public static void main(String args[])

{

//error: You can't create object of it

AbstractDemo obj = new AbstractDemo();

obj.anotherMethod();

}

}

Output:

Unresolved compilation problem: Cannot instantiate the type AbstractDemo

**Abstract method in Java with example**

A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well. In the last tutorial we discussed Abstract class.

This is how an abstract method looks in java:

abstract class Sum{

public abstract int myMethod(int n1, int n2);

}

**Rules of Abstract Method**

* Abstract methods don’t have body; they just have method signature as shown above.
* If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn’t need to have an abstract method compulsory.
* If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

**Example 1: abstract method in an abstract class**

//abstract class

abstract class Sum{

/\* These two are abstract methods, the child class

\* must implement these methods

\*/

public abstract int sumOfTwo(int n1, int n2);

public abstract int sumOfThree(int n1, int n2, int n3);

//Regular method

public void disp(){

System.out.println("Method of class Sum");

}

}

//Regular class extends abstract class

class Demo extends Sum{

/\* If I don't provide the implementation of these two methods, the

\* program will throw compilation error.

\*/

public int sumOfTwo(int num1, int num2){

return num1+num2;

}

public int sumOfThree(int num1, int num2, int num3){

return num1+num2+num3;

}

public static void main(String args[]){

Sum obj = new Demo();

System.out.println(obj.sumOfTwo(3, 7));

System.out.println(obj.sumOfThree(4, 3, 19));

obj.disp();

}

}

Output:

10

26

Method of class Sum

**Example 2: abstract method in interface**

All the methods of an interface are public abstract by default. You cannot have concrete (regular methods with body) methods in an interface.

//Interface

interface Multiply{

//abstract methods

public abstract int multiplyTwo(int n1, int n2);

/\* We need not to mention public and abstract in interface

\* as all the methods in interface are

\* public and abstract by default so the compiler will

\* treat this as

\* public abstract multiplyThree(int n1, int n2, int n3);

\*/

int multiplyThree(int n1, int n2, int n3);

/\* Regular (or concrete) methods are not allowed in an interface

\* so if I uncomment this method, you will get compilation error

\* public void disp(){

\* System.out.println("I will give error if u uncomment me");

\* }

\*/

}

class Demo implements Multiply{

public int multiplyTwo(int num1, int num2){

return num1\*num2;

}

public int multiplyThree(int num1, int num2, int num3){

return num1\*num2\*num3;

}

public static void main(String args[]){

Multiply obj = new Demo();

System.out.println(obj.multiplyTwo(3, 7));

System.out.println(obj.multiplyThree(1, 9, 0));

}

}

Output:

21

0

**Interface**

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

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 Also, the variables declared in an interface are public, static & final by default.

**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.

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 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

As discussed above, an interface cannot implement another interface. It has to extend the other interface. See the below example where we have two interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf2 as well as Inf1.

**Tag or Marker interface in Java**

An empty interface is known as tag or marker interface. For example, Serializable, EventListener, Remote(java.rmi.Remote) are tag interfaces. These interfaces do not have any field and methods in it.

**Nested Interface**

An interface which is declared inside another interface or class is called nested interface. They are also known as inner interface. For example, Entry interface in collections framework is declared inside Map interface, that’s why we don’ use it directly, rather we use it like this: Map. Entry

**Key points:** Here are the key points to remember about interfaces:  
1) We can’t instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

3) implements keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as private, protected or transient.

7) All the interface methods are by default **abstract and public**.

8) Variables declared in interface are **public, static and final** by default.

interface Try

{

int a=10;

public int a=10;

public static final int a=10;

final int a=10;

static int a=0;

}

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

interface Try

{

int x;//Compile-time error

}

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

class Sample implements Try

{

public static void main(String args[])

{

x=20; //compile time error

}

}

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A **class** can implement any **number of interfaces**.

13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

interface A

{

public void aaa();

}

interface B

{

public void aaa();

}

class Central implements A,B

{

public void aaa()

{

//Any Code here

}

public static void main(String args[])

{

//Statements

}

}

14) A class cannot implement two interfaces that have methods with same name but different return type.

interface A

{

public void aaa();

}

interface B

{

public int aaa();

}

class Central implements A,B

{

public void aaa() // error

{

}

public int aaa() // error

{

}

public static void main(String args[])

{

}

}

15) Variable names conflicts can be resolved by interface name.

interface A

{

int x=10;

}

interface B

{

int x=100;

}

class Hello implements A,B

{

public static void Main(String args[])

{

/\* reference to x is ambiguous both variables are x

\* so we are using interface name to resolve the

\* variable

\*/

System.out.println(x);

System.out.println(A.x);

System.out.println(B.x);

}

}

**Advantages of interface in Java**

Without bothering about the implementation part, we can achieve the security of implementation

In java, multiple inheritance is not allowed, however you can use interface to make use of it as you can implement more than one interface.

**Encapsulation**

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare class variables/attributes as private
* provide public **get** and **set** methods to access and update the value of a private variable

**Get and Set**

* private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.
* The get method returns the variable value, and the set method sets the value.
* Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

public class Person {

private String name; // private = restricted access

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}

}

The get method returns the value of the variable name.

The set method takes a parameter (newName) and assigns it to the name variable. This keyword is used to refer to the current object.

However, as the name variable is declared as private, we **cannot** access it from outside this class:

public class Main {

public static void main(String[] args) {

Person myObj = new Person();

myObj.name = "John"; // error

System.out.println(myObj.name); // error

}

}

If the variable was declared as public, we would expect the following output: John

However, as we try to access a private variable, we get an error:

MyClass.java:4: error: name has private access in Person  
    myObj.name = "John";  
         ^  
MyClass.java:5: error: name has private access in Person  
    System.out.println(myObj.name);  
                  ^  
2 errors

Instead, we use the getName() and setName() methods to acccess and update the variable:

public class Main {

public static void main(String[] args) {

Person myObj = new Person();

myObj.setName("John"); // Set the value of the name variable to "John"

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

}

}

// Outputs "John"

**Why Encapsulation.?**

* Hiding sensitive data from user
* Better control of class attributes and methods
* Class attributes can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
* Flexible: the programmer can change one part of the code without affecting other parts
* Increased security of data
* **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
* **Increased Flexibility:** We can make the variables of the class as read-only or write-know that only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
* **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
* **Testing code is easy:** Encapsulated code is easy to test for unit testing.