



# CORE JAVA

By : Kalpesh Chauhan



# OBJECT ORIENTED PROGRAMMING

# JAVA OOPS CONCEPTS

In this chapter, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **data binding**, **polymorphism**, etc.

**Simula** is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

**Smalltalk** is considered the first truly object-oriented programming language.

The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

The main aim of object-oriented programming is to implement real-world entities for example object, classes, abstraction, inheritance, polymorphism, etc.

# OOPS (OBJECT-ORIENTED PROGRAMMING SYSTEM)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

# OBJECT

Any entity that has state and behavior is known as an object. For example a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

**Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

# CLASS

*Collection of objects* is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

# INHERITANCE

*When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.*

# POLYMORPHISM

If *one task is performed by different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.



# ABSTRACTION

*Hiding internal details and showing functionality* is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

# ENCAPSULATION

*Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example capsule, it is wrapped with different medicines.*

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

# ADVANTAGE OF OOPS OVER PROCEDURE-ORIENTED PROGRAMMING LANGUAGE

- 1) OOPs makes development and maintenance easier whereas in a procedure-oriented programming language it is not easy to manage if code grows as project size increases.
- 2) OOPs provides data hiding whereas in a procedure-oriented programming language a global data can be accessed from anywhere.
- 3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.



# JAVA NAMING CONVENTIONS



Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.

But, it is not forced to follow. So, it is known as convention not rule.

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

# ADVANTAGE OF NAMING CONVENTIONS IN JAVA

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

# CAMELCASE IN JAVA NAMING CONVENTIONS

Java follows camelcase syntax for naming the class, interface, method and variable.

If name is combined with two words, second word will start with uppercase letter always e.g. `actionPerformed()`, `firstName`, `ActionEvent`, `ActionListener` etc.





# OBJECT AND CLASS

# OBJECTS AND CLASSES IN JAVA

In this chapter, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as logical entity whereas a class in Java is a logical entity only.

# WHAT IS AN OBJECT IN JAVA

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics.

**State:** represents the data (value) of an object.

**Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.

**Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

# Characteristics of Object

**A**

## State

Represents the data of an object.

## Behavior

represents the behavior of an object such as deposit, withdraw, etc.

**B**

**C**

## Identity

It is used internally by the JVM to identify each object uniquely.

# WHAT IS A CLASS IN JAVA

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

**Fields**

**Methods**

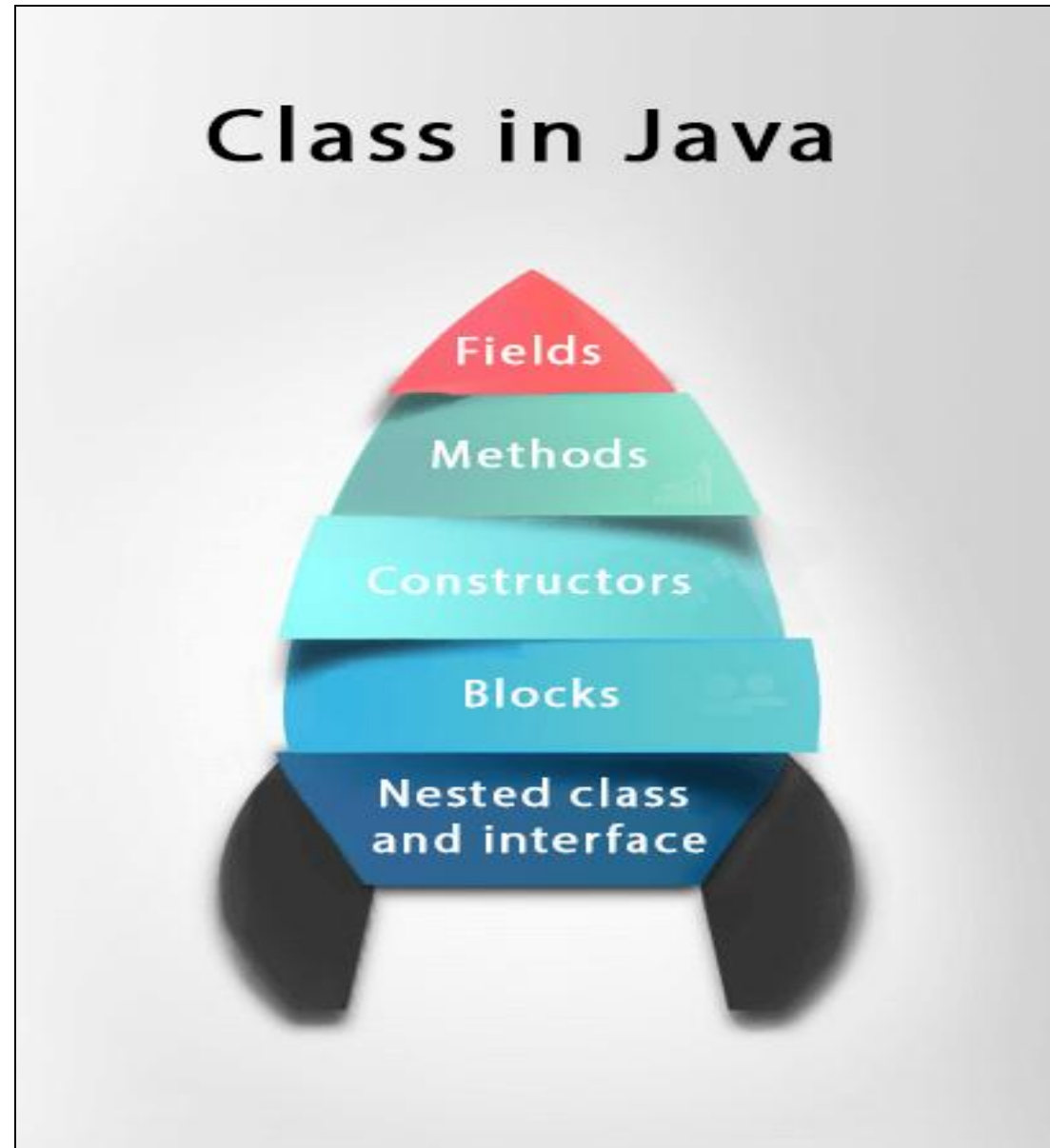
**Constructors**

**Blocks**

**Nested class and interface**

Syntax to declare a class:

```
class <class_name> {  
    field;  
    method;  
}
```



# INSTANCE VARIABLE IN JAVA

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.



# METHOD IN JAVA

In Java, a method is like a function which is used to expose the behavior of an object.

*Advantage of Method*

1. Code Reusability
2. Code Optimization

# NEW KEYWORD IN JAVA

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

# OBJECT AND CLASS EXAMPLE: MAIN WITHIN THE CLASS

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

[Click here to open example](#)

# OBJECT AND CLASS EXAMPLE: MAIN OUTSIDE THE CLASS

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different java files or single java file. If you define multiple classes in a single java source file, it is a good idea to save the file name with the class name which has main() method.

[Click here to open example](#)

# WAYS TO INITIALIZE OBJECT

There are 3 ways to initialize object in java.

1. By reference variable
2. By method
3. By constructor

# OBJECT AND CLASS EXAMPLE: INITIALIZATION THROUGH REFERENCE

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

[Click here to view example](#)

# OBJECT AND CLASS EXAMPLE: INITIALIZATION THROUGH METHOD

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

[Click here to view example](#)

# OBJECT AND CLASS EXAMPLE: INITIALIZATION THROUGH A CONSTRUCTOR

We will learn about constructors in next topic.





# CONSTRUCTOR

# CONSTRUCTORS IN JAVA

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the object is created, and memory is allocated for the object.

It is a special type of method which is used to initialize the object.

# WHEN IS A CONSTRUCTOR CALLED

Every time an object is created using `new()` keyword, at least one constructor is called. It calls a default constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

# RULES FOR CREATING JAVA CONSTRUCTOR

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

# TYPES OF JAVA CONSTRUCTORS

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

# JAVA DEFAULT CONSTRUCTOR

A constructor is called "Default Constructor" when it doesn't have any parameter.

**Syntax of default constructor:**

```
<class_name>() {  
    }  
}
```

***Rule: If there is no constructor in a class, compiler automatically creates a default constructor***

```
class Bike {  
}
```



Compiler



```
class Bike {  
    Bike (){}  
}
```

# JAVA PARAMETERIZED CONSTRUCTOR

A constructor which has a specific number of parameters is called a parameterized constructor.



# WHY USE THE PARAMETERIZED CONSTRUCTOR?

The parameterized constructor is used to provide different values to the distinct objects. However, you can provide the same values also.

# CONSTRUCTOR OVERLOADING IN JAVA

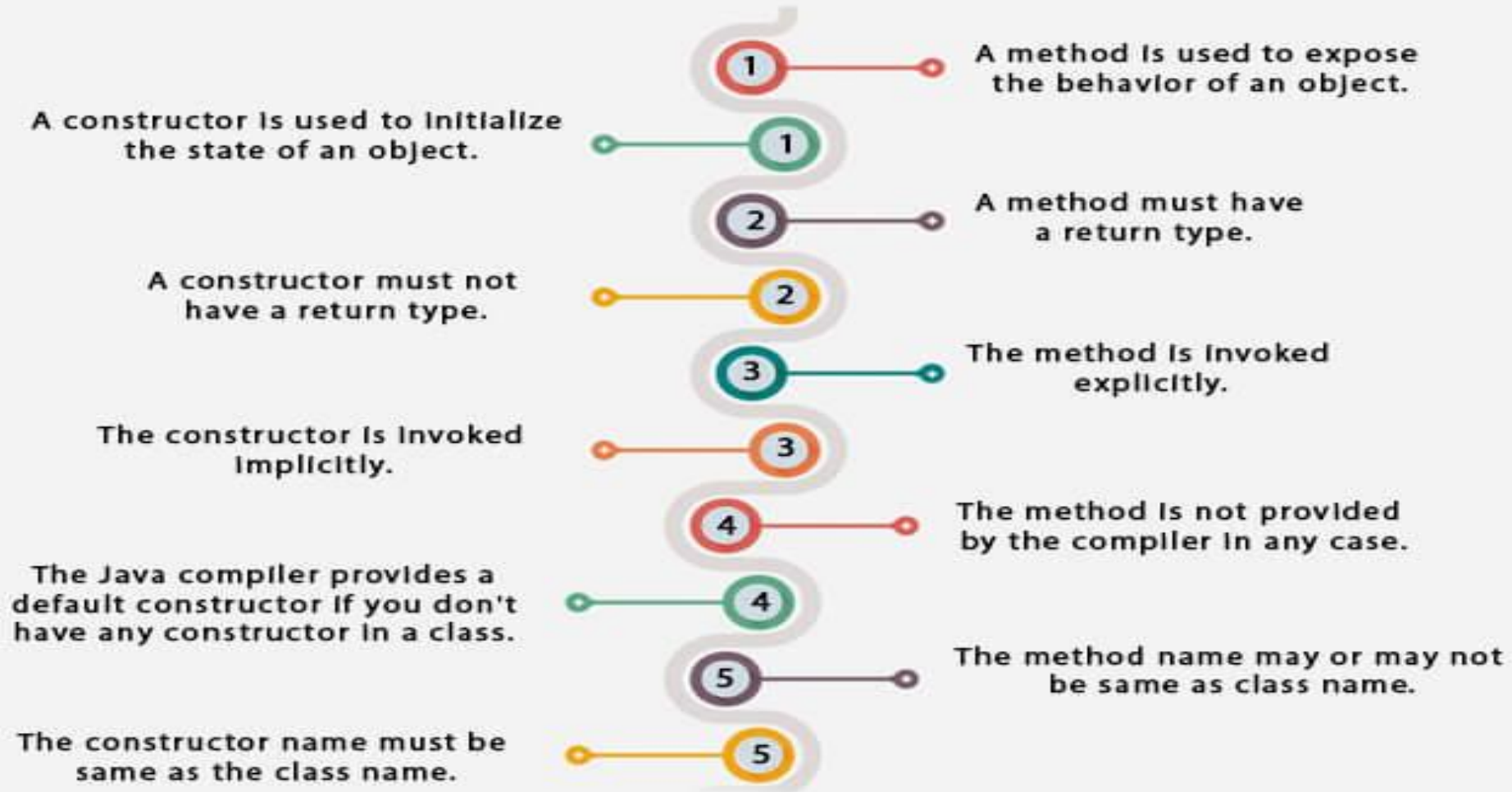
In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.



# DIFFERENCE BETWEEN CONSTRUCTOR AND METHOD IN JAVA

# Difference between constructor and method in Java



# JAVA COPY CONSTRUCTOR

There is no copy constructor in java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

1. By constructor
2. By assigning the values of one object into another
3. By clone() method of Object class



In this example, we are going to copy the values of one object into another using java constructor

[Click here to view example](#)

# CAN CONSTRUCTOR PERFORM OTHER TASKS INSTEAD OF INITIALIZATION?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.



# STATIC KEYWORD



# JAVA STATIC KEYWORD

The **static keyword** in Java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

# JAVA STATIC VARIABLE

If you declare any variable as static, it is known as a static variable.

The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

# *UNDERSTANDING THE PROBLEM WITHOUT STATIC VARIABLE*

```
class Student
```


```
{
```

```
    int rollno;
```

```
    String name;
```

```
    String college="Darshan Institute of Engineering Technology ";
```

```
}
```



Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

**Java static property is shared to all objects**

# JAVA STATIC METHOD

If you apply static keyword with any method, it is known as static method.

1. A static method belongs to the class rather than the object of a class.
2. A static method can be invoked without the need for creating an instance of a class.
3. A static method can access static data member and can change the value of it.


Example of static method

# RESTRICTIONS FOR THE STATIC METHOD

There are two main restrictions for the static method. They are:

The static method can not use non static data member or call non-static method directly.

this and super cannot be used in static context.



```
class A
{
    int a=40;//non static

    public static void main(String args[])
    {
        System.out.println(a);
    }
}
```

# WHY IS THE JAVA MAIN METHOD STATIC?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.



# JAVA STATIC BLOCK

Is used to initialize the static data member.

It is executed before the main method at the time of class loading.

# EXAMPLE OF STATIC BLOCK

```
class A2
{
    static{
        System.out.println("static block is invoked");
    }

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

# STATIC NESTED CLASS

A **static class** i.e. created inside a **class** is called **static nested class** in **java**. It cannot access non-**static** data members and methods. It can be accessed by outer **class** name.

It can access **static** data members of outer **class** including private.

# JAVA STATIC NESTED CLASS EXAMPLE WITH STATIC METHOD

If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
class TestOuter2{
    static int data=30;
    static class Inner{
        static void msg(){System.out.println("data is "+data);}
    }
    public static void main(String args[]){
        TestOuter2.Inner.msg();//no need to create the instance of static nested class
    }
}
```



# THIS KEYWORD IN JAVA



There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

# USAGE OF JAVA THIS KEYWORD

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.



# THIS: TO REFER CURRENT CLASS INSTANCE VARIABLE

This keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Example with out this keyword

# THIS: TO INVOKE CURRENT CLASS METHOD

You may invoke the method of the current class by using the `this` keyword. If you don't use the `this` keyword, compiler automatically adds `this` keyword while invoking the method.

# THIS() : TO INVOKE CURRENT CLASS CONSTRUCTOR

The `this()` constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

```
class A{  
    A(){  
        this(5);  
        System.out.println("hello a");  
    }  
    A(int x){  
        System.out.println(x);  
    }  
}
```

# THIS: TO PASS AS AN ARGUMENT IN THE METHOD

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

## Example

# THIS: TO PASS AS ARGUMENT IN THE CONSTRUCTOR CALL

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

## Example

# THIS KEYWORD CAN BE USED TO RETURN CURRENT CLASS INSTANCE

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
return_type method_name(){  
    return this;  
}
```


# EXAMPLE OF THIS KEYWORD THAT YOU RETURN AS A STATEMENT FROM THE METHOD

Example



# INHERITANCE IN JAVA





**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

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

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

# IS-A RELATIONSHIP:

In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

# HAS-A RELATIONSHIP:

Composition(HAS-A) simply mean the use of instance variables that are references to other objects. For example Maruti has Engine, or House has Bathroom.

# WHY USE INHERITANCE IN JAVA

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

# TERMS USED IN INHERITANCE

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

# THE SYNTAX OF JAVA INHERITANCE

```
class Subclass-name extends Superclass-name  
{  
    //methods and fields  
}
```

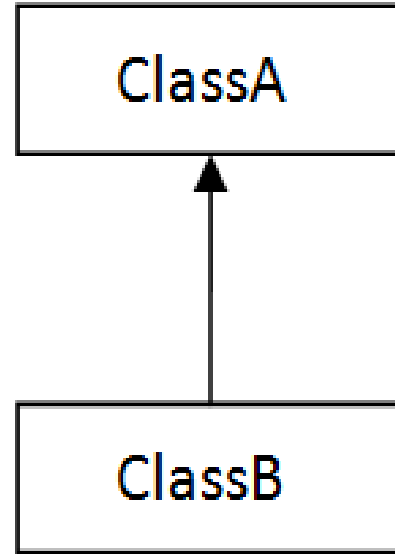
The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

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

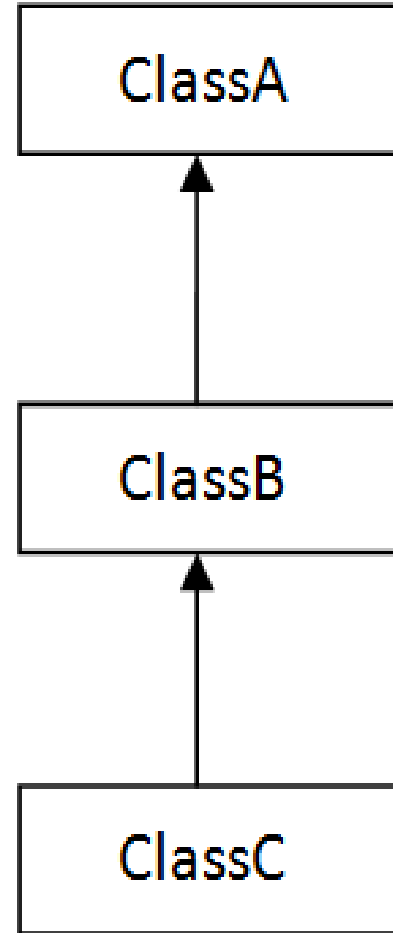
# TYPES OF INHERITANCE IN JAVA

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

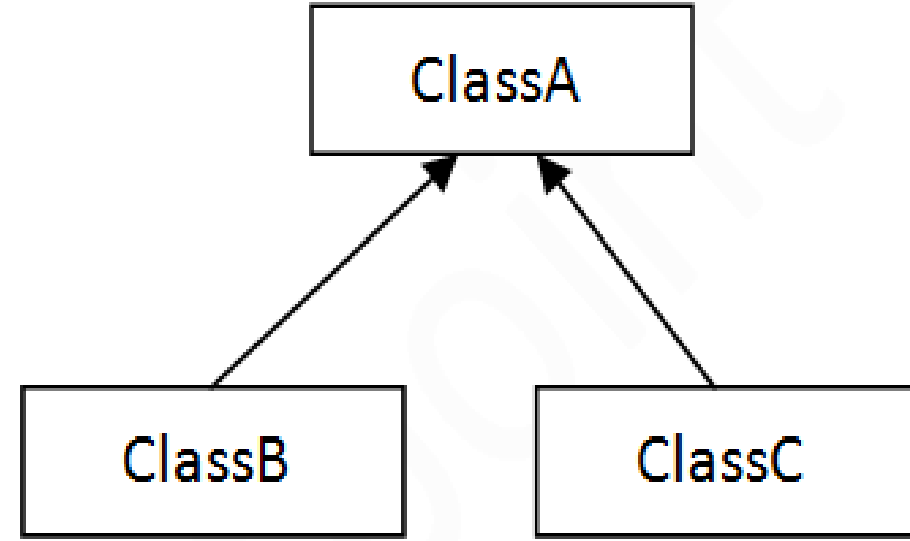
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



1) Single



2) Multilevel



3) Hierarchical



# WHY MULTIPLE INHERITANCE IS NOT SUPPORTED IN JAVA?

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

# AGGREGATION IN JAVA

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, email etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

# WHEN USE AGGREGATION?

Code reuse is also best achieved by aggregation when there is no is-a relationship.

Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.



# METHOD OVERLOADING IN JAVA



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

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as `a(int,int)` for two parameters, and `b(int,int,int)` for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

# ADVANTAGE OF METHOD OVERLOADING

Method overloading *increases the readability of the program.*

# DIFFERENT WAYS TO OVERLOAD THE METHOD

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

# METHOD OVERLOADING: CHANGING NO. OF ARGUMENTS

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

[example](#)



# METHOD OVERLOADING: CHANGING DATA TYPE OF ARGUMENTS

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

## Example

# METHOD OVERRIDING IN JAVA

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.

Usage of Java Method Overriding

1. Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
2. Method overriding is used for runtime polymorphism

# Rules for Java Method Overriding



Method must have same name as in the parent class

STEP  
01

STEP  
02

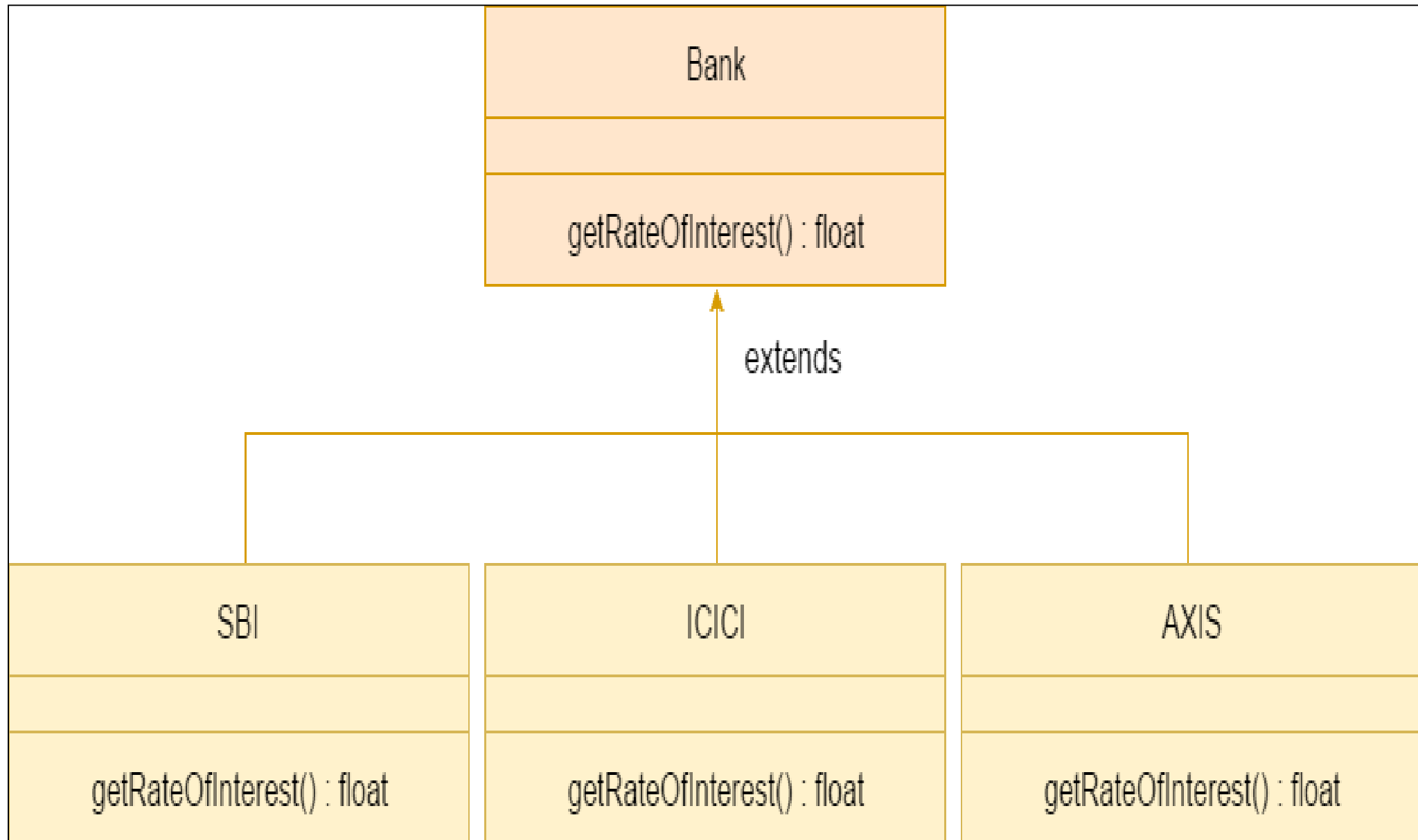
Method must have same parameter as in the parent class.

There must be IS-A relationship (inheritance).

STEP  
03

# A REAL EXAMPLE OF JAVA METHOD OVERRIDING

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



# CAN WE OVERRIDE STATIC METHOD?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

# WHY CAN WE NOT OVERRIDE STATIC METHOD?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

# CAN WE OVERRIDE JAVA MAIN METHOD?

No, because the main is a static method.



# COVARIANT RETURN TYPE

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

[example](#)



# SUPER KEYWORD IN JAVA



The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

# Usage of Super Keyword

**1**

**Super can be used to refer immediate parent class instance variable.**

**2**

**Super can be used to invoke immediate parent class method.**

**3**

**super() can be used to invoke immediate parent class constructor.**



# INSTANCE INITIALIZER BLOCK



**Instance Initializer block** is used to initialize the instance data member. It runs each time when an object of the class is created.

The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

What is the use of instance initializer block while we can directly assign a value in instance data member?

```
class Bike
{
    int speed=100;
}
```


# WHY USE INSTANCE INITIALIZER BLOCK?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.



# WHAT IS INVOKED FIRST, INSTANCE INITIALIZER BLOCK OR CONSTRUCTOR?

```
class Bike8{  
    int speed;  
    Bike8(){  
        System.out.println("constructor is invoked");    }  
    {        System.out.println("instance initializer block invoked"); }  
    public static void main(String args[]){  
        Bike8 b1=new Bike8();  
        Bike8 b2=new Bike8();  
    }  
}
```



In the above example, it seems that instance initializer block is firstly invoked but NO. Instance initializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement `super()`. So firstly, constructor is invoked.

Note: The java compiler copies the code of instance initializer block in every constructor.

# RULES FOR INSTANCE INITIALIZER BLOCK :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after `super()` constructor call).
3. The instance initializer block comes in the order in which they appear.




# FINAL KEYWORD IN JAVA



The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class



The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

# JAVA FINAL VARIABLE

If you make any variable as final, you cannot change the value of final variable(It will be constant).

# JAVA FINAL METHOD

If you make any method as final, you cannot override it.




# JAVA FINAL CLASS

If you make any class as final, you cannot extend it.



# POLYMORPHISM IN JAVA



**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 you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

# RUNTIME POLYMORPHISM IN JAVA

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

# UPCASTING

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

```
class A
```

```
{
```

```
}
```

```
class B extends A{
```

```
A a=new B();//upcasting
```

# EXAMPLE OF JAVA RUNTIME POLYMORPHISM

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
    void run(){
        System.out.println("running");
    }
}

class Splendor extends Bike{
    void run(){
        System.out.println("running safely with 60km");
    }
}

public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
}
}
```

# JAVA RUNTIME POLYMORPHISM EXAMPLE: BANK

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



```
class Bank{  
float getRateOfInterest(){return 0;}  
}  
class SBI extends Bank{  
float getRateOfInterest(){return 8.4f;}  
}  
class ICICI extends Bank{  
float getRateOfInterest(){return 7.3f;}  
}  
class AXIS extends Bank{  
float getRateOfInterest(){return 9.7f;}  
}
```

```
class TestPolymorphism{  
public static void main(String args[]){  
    Bank b;  
    b=new SBI();  
    System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());  
    b=new ICICI();  
    System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());  
    b=new AXIS();  
    System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());  
}  
}
```



# STATIC BINDING AND DYNAMIC BINDING



Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

# UNDERSTANDING TYPE

Let's understand the type of instance.

1. Variable
2. Instance
3. object

# VARIABLES HAVE A TYPE

Each variable has a type, it may be primitive and non-primitive.

```
int data=30;
```

Here data variable is a type of int.

# REFERENCES HAVE A TYPE

```
class Dog{  
    public static void main(String args[]){  
        Dog d1;//Here d1 is a type of Dog  
    }  
}
```

# OBJECTS HAVE A TYPE

An object is an instance of particular java class, but it is also an instance of its superclass.



```
class Animal{  
}
```

```
class Dog extends Animal{  
    public static void main(String args[]) {  
        Dog d1=new Dog();  
    }  
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

# STATIC BINDING

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

```
class Dog{  
    private void eat(){  
        System.out.println("dog is eating...");  
    }  
  
    public static void main(String args[]){  
        Dog d1=new Dog();  
        d1.eat();  
    }  
}
```

# DYNAMIC BINDING

When type of the object is determined at run-time, it is known as dynamic binding.

```
class Animal{  
    void eat(){  
        System.out.println("animal is eating...");  
    }  
}
```

```
class Dog extends Animal{  
    void eat(){  
        System.out.println("dog is eating...");  
    }  
}
```

```
public static void main(String args[]){  
    Animal a=new Dog();  
    a.eat();  
}
```

# JAVA INSTANCEOF

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as *type comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

# SIMPLE EXAMPLE OF JAVA INSTANCEOF

Let's see the simple example of instance operator where it tests the current class.

```
class Simple1{  
    public static void main(String args[]){  
        Simple1 s=new Simple1();  
        System.out.println(s instanceof Simple1);//true  
    }  
}
```

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.


# ANOTHER EXAMPLE OF JAVA INSTANCEOF OPERATOR

```
class Animal{}  
class Dog1 extends Animal{//Dog inherits Animal  
    public static void main(String args[]){  
        Dog1 d=new Dog1();  
        System.out.println(d instanceof Animal);//true  
    }  
}
```



# DOWNCASTING WITH JAVA INSTANCEOF OPERATOR

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, `ClassCastException` is thrown at runtime. But if we use `instanceof` operator, downcasting is possible.



```
Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, `ClassCastException` is thrown at runtime.

```
Dog d=(Dog)new Animal();
```

```
//Compiles successfully but ClassCastException is thrown at runtime
```

# POSSIBILITY OF DOWNCASTING WITH INSTANCEOF

Let's see the example, where downcasting is possible by instanceof operator.

```
class Animal {  
}
```

```
class Dog3 extends Animal {  
    static void method(Animal a) {  
        if(a instanceof Dog3){  
            Dog3 d=(Dog3)a;//downcasting  
            System.out.println("ok downcasting performed");  
        }  
    }  
}
```

```
public static void main (String [] args) {  
    Animal a=new Dog3();  
    Dog3.method(a);  
}  
}
```



# ABSTRACT CLASS IN JAVA



A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

# ABSTRACTION IN JAVA

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

# WAYS TO ACHIEVE ABSTRACTION

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)



# ABSTRACT CLASS IN JAVA

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

## Rules for Java Abstract class



1

An abstract class must be declared with an abstract keyword.

2

It can have abstract and non-abstract methods.

3

It cannot be instantiated.

4

It can have final methods

5

It can have constructors and static methods also.

# ABSTRACT METHOD IN JAVA

A method which is declared as abstract and does not have implementation is known as an abstract method.

## **Example of abstract method**

**abstract void** printStatus();//no method body and abstract

# EXAMPLE OF ABSTRACT CLASS THAT HAS AN ABSTRACT METHOD



In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class

```
abstract class Bike{  
    abstract void run();  
}  
  
class Honda4 extends Bike{  
    void run(){  
        System.out.println("running safely");  
    }  
  
    public static void main(String args[]){  
        Bike obj = new Honda4();  
        obj.run();  
    }  
}
```

# UNDERSTANDING THE REAL SCENARIO OF ABSTRACT CLASS

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

```
abstract class Shape{
abstract void draw();
}           //In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}           //In real scenario, method is called by programmer or user
class TestAbstraction1{
    public static void main(String args[]){
        Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape()
        s.draw();
    }
}
```



# ABSTRACT CLASS HAVING CONSTRUCTOR, DATA MEMBER AND METHODS

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

```
abstract class Bike{  
    Bike(){  
        System.out.println("bike is created");  
    }  
  
    abstract void run();  
    void changeGear(){  
        System.out.println("gear changed");  
    }  
}
```

//Creating a Child class which inherits Abstract class

```
class Honda extends Bike{  
    void run(){  
        System.out.println("running safely..");  
    }  
}
```

//Creating a Test class which calls abstract and non-abstract methods

```
class TestAbstraction2{  
    public static void main(String args[]){  
        Bike obj = new Honda();  
        obj.run();  
        obj.changeGear();  
    }  
}
```



# INTERFACE IN JAVA



An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a *mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.



Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

# WHY USE JAVA INTERFACE?

There are mainly three reasons to use interface. They are given below.

**It is used to achieve abstraction.**

**1**

**2**

**By interface, we can support the functionality of multiple inheritance.**

**It can be used to achieve loose coupling.**

**3**

# HOW TO DECLARE AN INTERFACE?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```



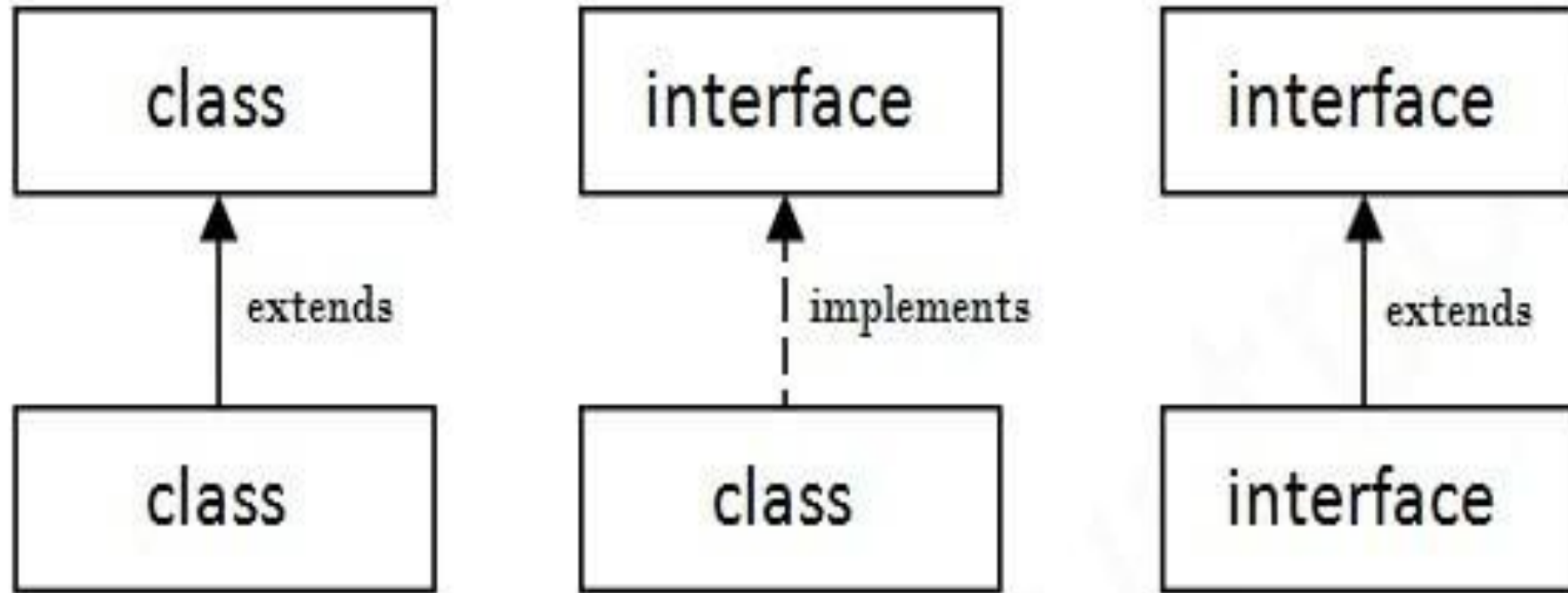
# INTERNAL ADDITION BY THE COMPILER

*The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.*

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

# THE RELATIONSHIP BETWEEN CLASSES AND INTERFACES

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



# DIFFERENCE BETWEEN ABSTRACT CLASS AND INTERFACE

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
<b>Abstract class can have abstract and non-abstract methods.</b>	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
<b>Abstract class doesn't support multiple inheritance.</b>	Interface supports multiple inheritance.
<b>Abstract class can have final, non-final, static and non-static variables.</b>	Interface has only static and final variables.
<b>Abstract class can provide the implementation of interface.</b>	Interface can't provide the implementation of abstract class.
<b>The abstract keyword is used to declare abstract class.</b>	The interface keyword is used to declare interface.
<b>An abstract class can extend another Java class and implement multiple Java interfaces.</b>	An interface can extend another Java interface only.

Abstract class	Interface
7) An abstract class can be extended using keyword "extends".	An interface class can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).



# JAVA PACKAGE



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

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

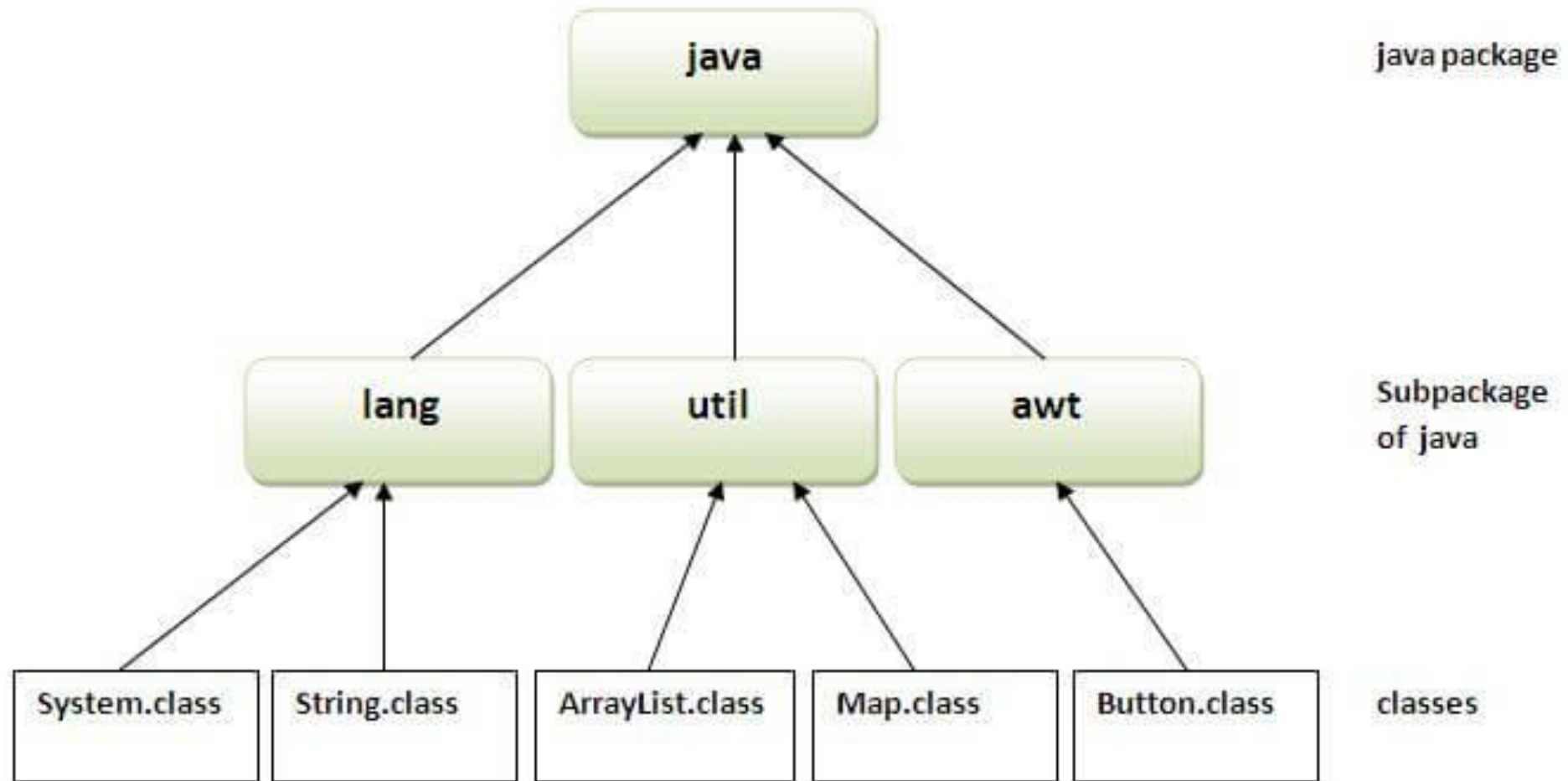
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

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



# ADVANTAGE OF JAVA PACKAGE

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision



# SIMPLE EXAMPLE OF JAVA PACKAGE

The **package keyword** is used to create a package in java.

//save as Simple.java

```
package mypack;  
  
public class Simple{  
    public static void main(String args[]){  
        System.out.println("Welcome to package");  
    }  
}
```

# HOW TO COMPILE JAVA PACKAGE

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

# HOW TO RUN JAVA PACKAGE PROGRAM

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

**To Compile:** `javac -d . Simple.java`

**To Run:** `java mypack.Simple`

# HOW TO ACCESS PACKAGE FROM ANOTHER PACKAGE?

There are three ways to access the package from outside the package.

1. `import package.*;`
2. `import package.classname;`
3. fully qualified name.

# USING PACKAGENAME.\*

If you use `package.*` then all the classes and interfaces of this package will be accessible but not subpackages.

The `import` keyword is used to make the classes and interface of another package accessible to the current package.

# USING PACKAGENAME.CLASSNAME

If you import `package.classname` then only declared class of this package will be accessible.



# USING FULLY QUALIFIED NAME

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. `java.util` and `java.sql` packages contain `Date` class.

***Note: If you import a package, subpackages will not be imported.***

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

# SUBPACKAGE IN JAVA

Package inside the package is called the **subpackage**. It should be created to **categorize the package further**.

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

***The standard of defining package is domain.company.package e.g. com.java.bean or org.sssit.dao.***


# ACCESS MODIFIERS IN JAVA

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public



There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.



## **private access modifier**

The private access modifier is accessible only within class.

## **default access modifier**

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package.

## **protected access modifier**

The **protected access modifier** is accessible within package and outside the package but through inheritance only. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.



## **public access modifier**

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

# UNDERSTANDING ALL JAVA ACCESS MODIFIERS

Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Yes	No	No	No
Default	Yes	Yes	No	No
Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes

# ENCAPSULATION IN JAVA

**Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.



The **Java Bean** class is the example of a fully encapsulated class.



# ADVANTAGE OF ENCAPSULATION IN JAVA

By providing only a setter or getter method, you can make the class **read-only** or **write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### Example



**CONTINUE IN NEXT UNIT.....**