Java Oops

Why Is Java Popular?

As we know java is a high level, object-oriented programming language. One of the most important features of java is that it is platform independent. Java gets more popularity because of this feature.

Platform: An environment in which a program can be executed is known as a Platform. Platform can be hardware or software or may be both.

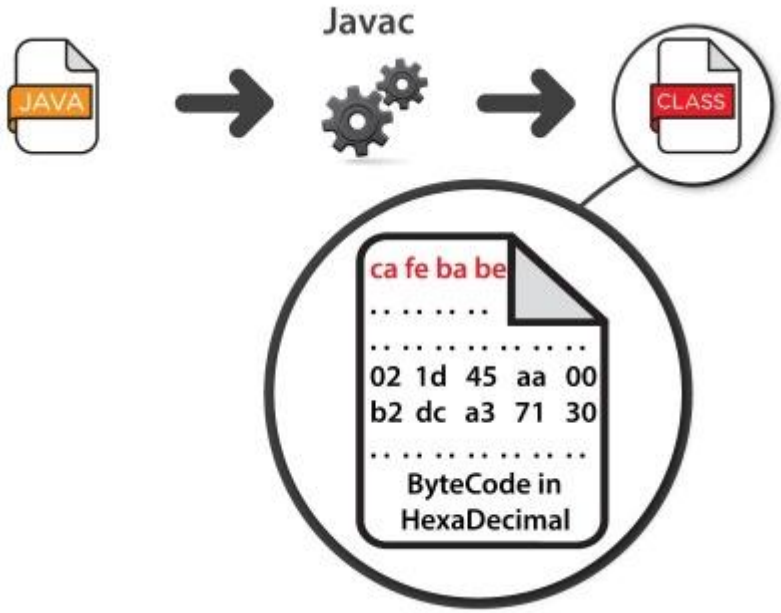
- Hardware platform means any physical machine. Which is used for executing the code.
- Software is a set of instructions which is developed for executing a particular task which is installed in a physical machine.

Platform independent means that java code does not depend on any machine. Basically here machine word is used for Operating System.

For Example: If you have written Java code on one machine, suppose Windows, then it can also be executed on Linux, Unix, Macintosh platform without making any changes. But this is not the case with languages like C and C++, these languages are platform dependent. It is not required to make any changes in code if you are executing your same Java code on a different operating system. That's why we can say java code is written once and run anywhere.

JVM stands for Java Virtual Machine which is platform dependent. Every operating system has its own JVM which has its own implementation of Java as per operating system. JVM is a vital part of java, it takes byte code as an input and converts it into machine code.

The byte code is generated by the compiler (JavaC). Java compiler compiles the java source code and converts it into bytecode after that code is handed over to JVM.



What makes Java different from some other programming languages?

Java is:

- Simple
- Distributed
- Multithreaded
- Object-oriented
- Platform independent
- Robust
- Secured
- Portable

Does Java Use Pointer Or Restricted Pointer?

C and C++ use pointers but java does not use pointers. Actually java has pointers only for internal work and it did not provide access to pointers to external users because of security, java users can't use

pointers in java programs. Java has an internal pointer which is used only by java, this feature is called restricted pointer.

Why Does Java Use an Internal Pointer?

Array internally uses a pointer. Java uses Array and array takes continuous memory storage. We can't move one element of Array to the next element without a pointer because we need the address of the next element and this address is provided by the pointer.

Difference between Compiler and Interpreter

Compilers and interpreters are programs that help convert the high level language (Source Code) into machine codes to be understood by the computers. A high level language is one that can be understood by humans. . However, computers cannot understand high level languages as we humans do. They can only understand the programs that are developed in binary systems known as machine code. To start with, a computer program is usually written in high level language described as a source code. These source codes must be converted into machine language and here comes the role of compilers and interpreters.

To start with, a compiler creates the program. It will analyse all the language statements to check if they are correct. If it comes across something incorrect, it will give an error message. If there are no errors spotted, the compiler will convert the source code into machine code. The compiler links the different code files into programs that can be run such as exe. Finally the program runs.

An interpreter creates the program. It neither links the files nor generated machine code. The source statements are executed line by line while executing the program.

Interpreter	**Compiler**

Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No Object Code is generated, hence are memory efficient.	Generates Object Code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

Figure: Compiler

Figure: Interpreter

Does Java Use Compiler Or Interpreter?

Java uses both compiler as well as interpreter while other programming languages use only one, that may be compiler or may be an interpreter.For Example C and C++ use only compiler to compile the code and python uses only interpreter. Java uses both Compiler and interpreter, it first uses compiler and then interpreter.

Object Oriented Programming

OOP concept revolves around real-world objects and their usages. Thus in OOP, the Hotel Management System can be thought of in terms of the objects involved. Now the question arises, What are objects? So objects are the real-world entity around which our system revolves or in other words system is dependent on objects. In the hotel management system what all objects you can think of in a hotel?

– “Room” which can represents the rooms in that hotel

– “Customer” which represents the customer who will visit to the hotel to book room

– “Booking” which can represents the booking of the room which will be done by customer

So, all these objects represent real world entities and make our system similar to real world entities. Thus you can say that the Object-Oriented programming concept views problems in terms of real-world objects.

Now, to describe or represent real-world objects or entities you need Classes. Class is used to bind characteristics(variables) and functionality. When an object is created which belongs to that class, that object has all the characteristics and methods which were defined in the class and has their specific value according to that object.

For example, in the hotel management system, to represent a room object in terms of class you have to define the properties of the room like room_number, room rent etc as well as methods (functions related to it) which are specific to that room only, for example every room in the hotel has CheckIn and CheckOut functionality of its own.

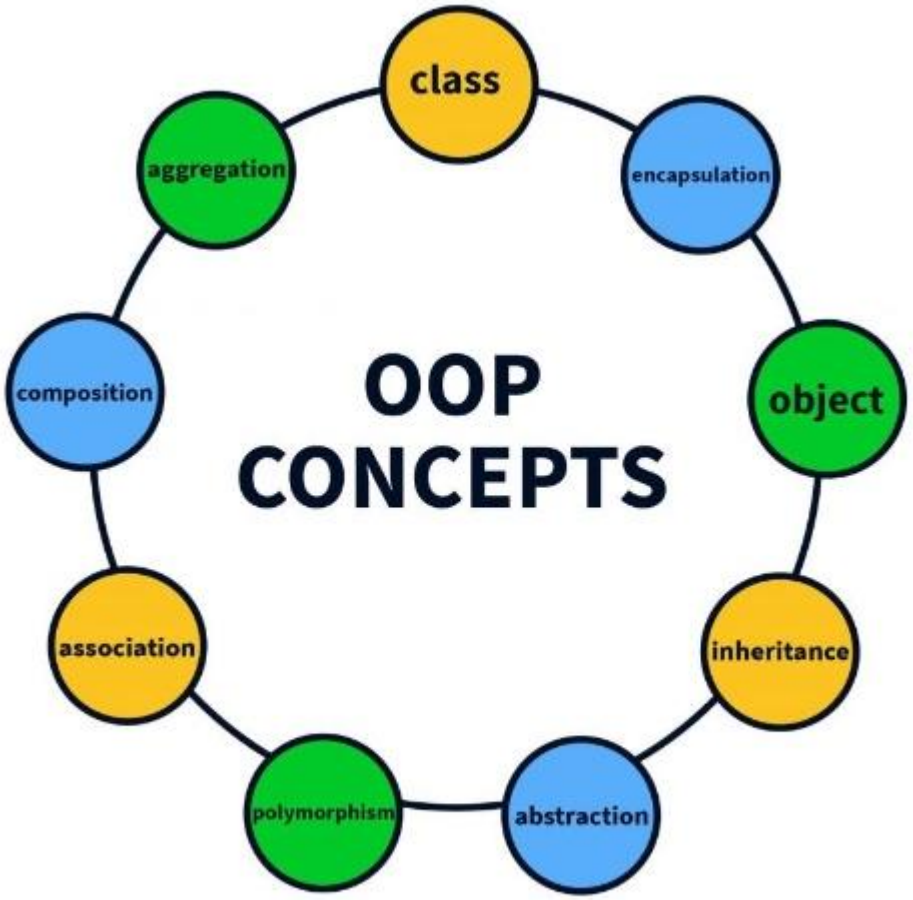
So every room will have some common characteristic properties(which could be number, rent, size, Ac/Non Ac, status) but the values associated (such as 102 room number, 2000 Rs per night, XL size, AC room, Occupied) to these properties will change for every room independently. Such characteristics are referred to as properties of the object, and their values as state.

Similar to the properties, every room will also have some functionalities which will help the system to access, fetch and update the values of the properties of the room. Such as, if a room has Status value equal to ‘Vacant’ and a new guest arrives and occupies that empty room.

The system must update the status of the room from ‘Vacant’ to ‘Occupied’. So, to perform this update, the room must have a functionality, say, ‘toggleStatus’. The functionality of this function will be switching the status of the room between ‘Vacant’ and ‘Occupied’. All these types of functionalities (which update and fetch the characteristics values) are known as methods of the object.

So now concluding a ‘Room’ class will be a blueprint containing the properties and the methods, but the class will not have any state but whenever its object is created, the object’s properties will have some values and contain all the functions described in blueprint(class). You can say that class represents a group of the same or similar type of objects. So, the “Room” class in the hotel management system is used to represent all the rooms of a hotel.

OOP concepts:



These are basic concepts or characteristics of OOPS. We will take a deeper look at each one:

Class A class can be considered as a blueprint or template for creating similar types of objects. Classes are a user-defined data type that defines two things:

- Attributes or data
- Behaviour or methods

Individual objects are then created using the class or blueprint.

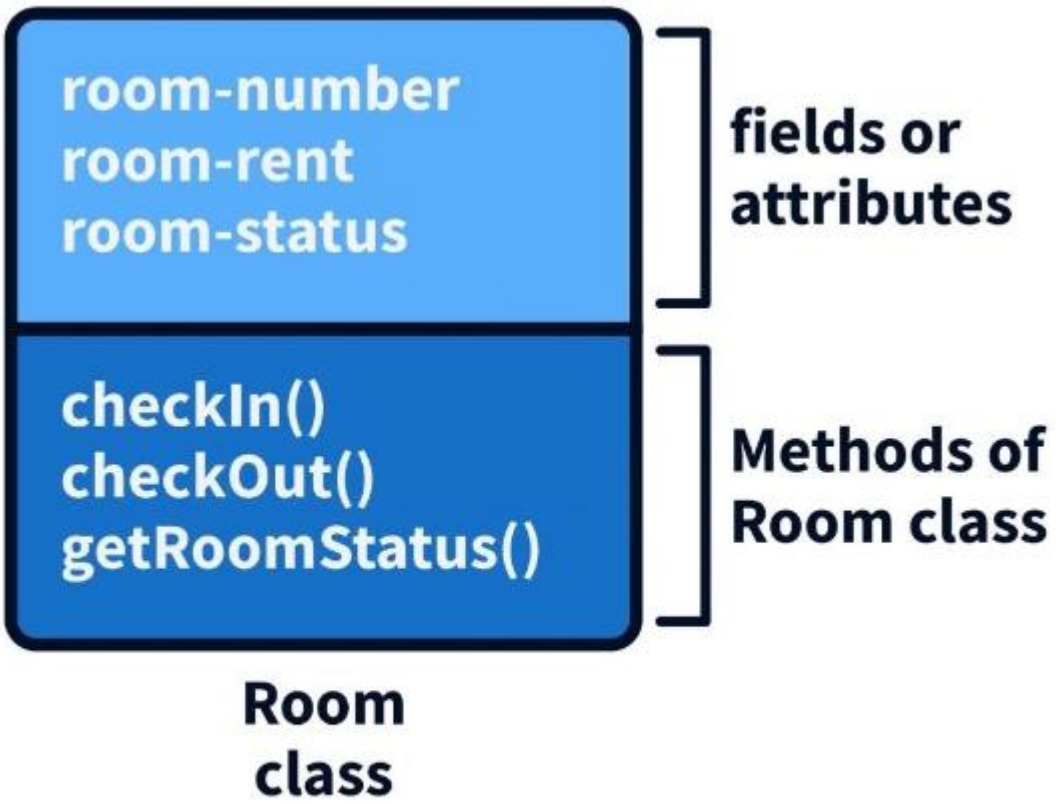
For example, in the hotel management system mentioned above, you can create a class “Room” to define each room in the hotel which will have attributes and behaviour of each room. So room class can contain the following fields or attributes:

– room_number
– room_rent
– room_status

And “Room” class can contain following methods to use above given fields:

– checkIn() : This function will be used to checkin in the room.
– checkout() : This function will be used to checkout of the room.
– getRoomStatus() : This function will give the status of the room whether it is occupied or not.

So in a “Room” class, here class is a template to represent a room in the hotel and you have to create or instantiate the Room class to represent an individual real-world thing that is a room of the hotel.



For example, we have defined the “Room” class to represent a hotel room. So to create an instance or object of the “Room” class you have to define an object for the “Room” class which represents a specific room of the hotel. In C++ to define object syntax is as follows:

```
class_name object_name;
```

Suppose you have to define room r1, so r1 will contain all the information specific to that room in the hotel. So to define room r1 for the hotel you can instantiate the “Room” class or create the object of Room class like this:

```
Room r1;
```

So, here r1 is an instance or object of class Room and will have its specific room_number, room_rent, room_status.

Note: Whenever you define a class, only the properties of the object are defined, no memory is allocated. So to allocate memory and to use data and functions of a class you have to create objects of that class.

Methods and Attributes

Methods: Methods are functions that are defined in class body or definition and they can perform various actions like updating the field or data of the object of the class, showing the data of the object of the class, etc. Methods generally specify the behaviour of a class. When the objects of a class are created, these objects can call the methods of the class. In the above example of Room class,

– checkIn()
– checkOut()
– getRoomStatus()

These are the methods that are defined in the class and these can be called using the objects of class “Room”.

Methods usually promote the reusability of code. Suppose, there is a piece of code that is used in code many times, so instead of writing the same code again and again you can put that code into methods and you can use that method. It also helps programmers to debug the code easily. The programmer has to check errors only at one place that is in the method instead of checking the whole code.

Attributes : Attributes are the data fields or information that are specific to an object. When an object of a class is created all the information specific to that object is stored in attributes or fields. In the above example of Room class,

– room_number
– room_rent
– room_status

These are attributes defined in the “Room” class. Suppose you create object r1 of Room class, then you can store information related to that particular room r1 in the attributes mentioned above.

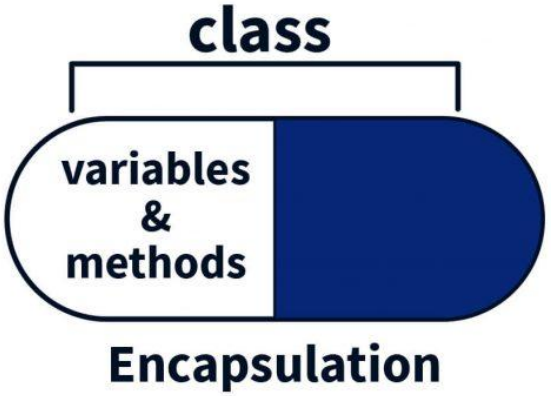
Note: You can use the “.” operator to access the attributes of objects or you can use getters and setters for objects.

Encapsulation

Encapsulation is a mechanism in which attributes and behaviours are bound together in a class. You can think of Encapsulation as just like a medicine capsule in which many medicines are combined or bound together. Similarly, encapsulation binds data and methods in a class

together. Also, Encapsulation provides a function of a secure layer by hiding the software or product internal implementation of code and internal data of the class and exposing only required information to the outer world.

Methods and data are defined in the class definition. Whenever an object of that class will be created, the data and methods are encapsulated or combined in that object or capsule. Just like in our “Room” class of “Hotel Management System” data and methods are bind together in one object.



In Encapsulation, the data of a class is inaccessible for any other outer classes or outer world and only the members of the same class can access that data. In encapsulation, the programmer requires to define some fields or data as public or private:

– Public: These methods or variables(fields) are accessible both from the same class as well as from outer classes also.

– Private: These methods or variables(fields) are accessible only in the same class.

You can take an example: Suppose in the above example of “Hotel Management System” you only want room_rent variable of “Room” class to be accessible only in that Room class or want that no method

from any other class can change the rent of that particular room, then we can declare that room_rent field to be “private” so that no one outside the class “Room” can change its value.

```
class Room {
  Private int room_rent;
};
```

Also here we are making the room_rent variable or field private so it can only be changed by the function of the same class. As we are hiding data from outer classes or the world, Encapsulation is also known as Data hiding.

Access Modifiers

In Java, access modifiers are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods.

Types of Access Modifier

There are four access modifiers keywords in Java and they are:

Modifier	**Description**
Default	declarations are visible only within the package (package private)
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Default Access Modifier

If we do not explicitly specify any access modifier for classes, methods, variables, etc, then by default the default access modifier is considered. For example,

```
package defaultPackage;
class Logger {
    void message(){
        System.out.println("This is a message");
    }
}
```

Here, the Logger class has the default access modifier. And the class is visible to all the classes that belong to the defaultPackage package. However, if we try to use the Loggerclass in another class outside of defaultPackage, we will get a compilation error.

Private Access Modifier

When variables and methods are declared private, they cannot be accessed outside of the class. For example,

```
class Data {
    // private variable
    private String name;
}

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

        // create an object of Data
        Data d = new Data();

        // access private variable and field from another class
        d.name = "Programiz";
    }
```

```
}
```

In the above example, we have declared a private variable named name. When we run the program, we will get the following error:

```
Main.java:18: error: name has private access in Data
        d.name = "Programiz";
         ^
```

The error is generated because we are trying to access the private variable of the Dataclass from the Main class.

You might be wondering what if we need to access those private variables. In this case, we can use the getters and setters method. For example,

```
class Data {
    private String name;

    // getter method
    public String getName() {
        return this.name;
    }
    // setter method
    public void setName(String name) {
        this.name= name;
    }
}
public class Main {
    public static void main(String[] main){
        Data d = new Data();
```

```
        // access the private variable using the getter and setter
        d.setName("Programiz");
        System.out.println(d.getName());
    }
}
```

Output:

```
The name is Programiz
```

In the above example, we have a private variable named name. In order to access the variable from the outer class, we have used methods: getName() and setName(). These methods are called getter and setter in Java.

Here, we have used the setter method (setName()) to assign value to the variable and the getter method (getName()) to access the variable.

We have used this keyword inside the setName() to refer to the variable of the class. To learn more on this keyword, visit Java this Keyword.

Protected Access Modifier

When methods and data members are declared protected, we can access them within the same package as well as from subclasses. For example,

```
class Animal {
    // protected method
    protected void display() {
        System.out.println("I am an animal");
    }
```

```
}

class Dog extends Animal {
    public static void main(String[] args) {

        // create an object of Dog class
        Dog dog = new Dog();
         // access protected method
        dog.display();
    }
}
```

Output:

```
I am an animal
```

In the above example, we have a protected method named display() inside the Animalclass. The Animal class is inherited by the Dog class. To learn more about inheritance, visit Java Inheritance.

We then created an object dog of the Dog class. Using the object we tried to access the protected method of the parent class.

Since protected methods can be accessed from the child classes, we are able to access the method of Animal class from the Dog class.

Public Access Modifier

When methods, variables, classes, and so on are declared public, then we can access them from anywhere. The public access modifier has no scope restriction. For example,

```
// Animal.java file
// public class
public class Animal {
    // public variable
    public int legCount;
```

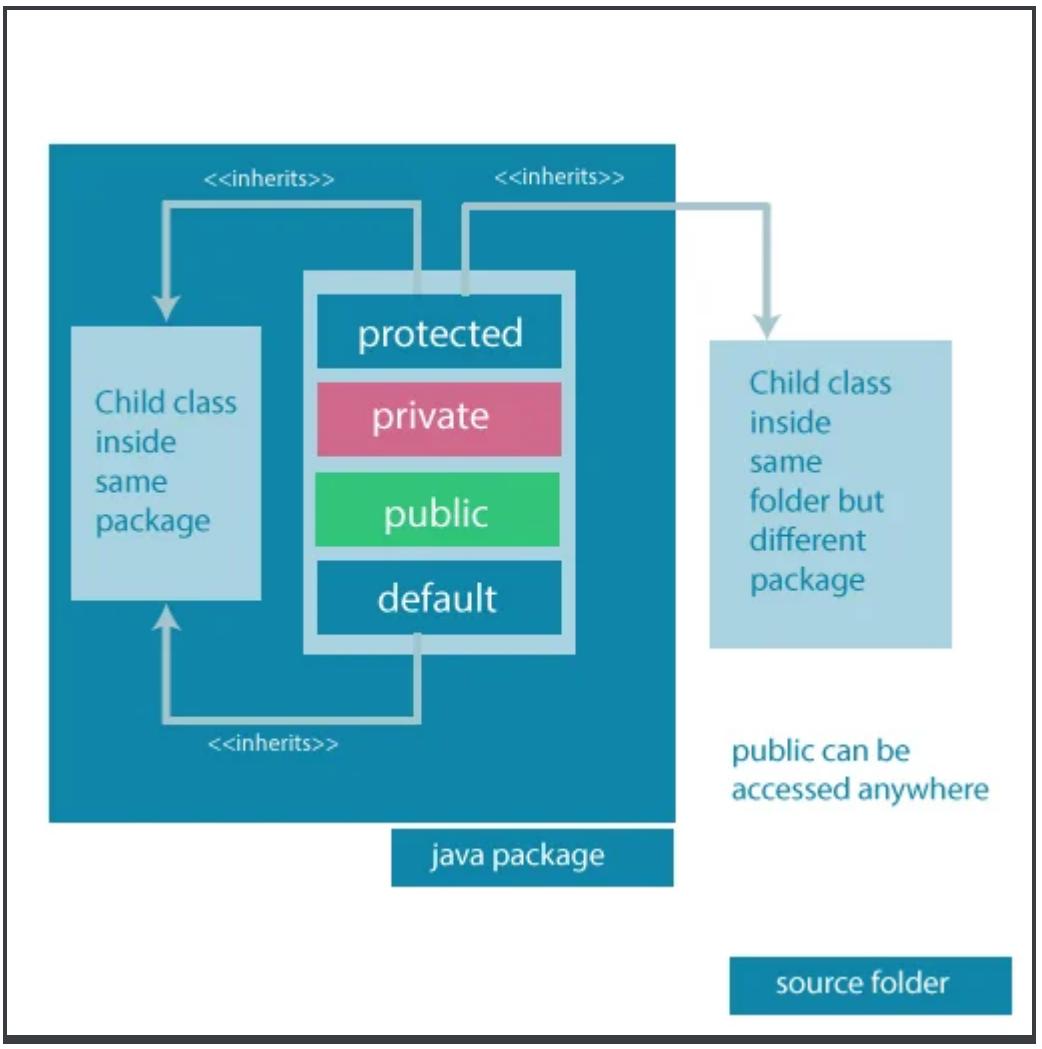
```
    // public method
    public void display() {
        System.out.println("I am an animal.");
        System.out.println("I have " + legCount + " legs.");
    }
}

// Main.java
public class Main {
    public static void main( String[] args ) {
        // accessing the public class
        Animal animal = new Animal();

        // accessing the public variable
        animal.legCount = 4;
        // accessing the public method
        animal.display();
    }
}
```

Output:

```
I am an animal.
I have 4 legs.
```

Here,

- The public class Animal is accessed from the Main class.
- The public variable legCount is accessed from the Main class.
- The public method display() is accessed from the Main class.

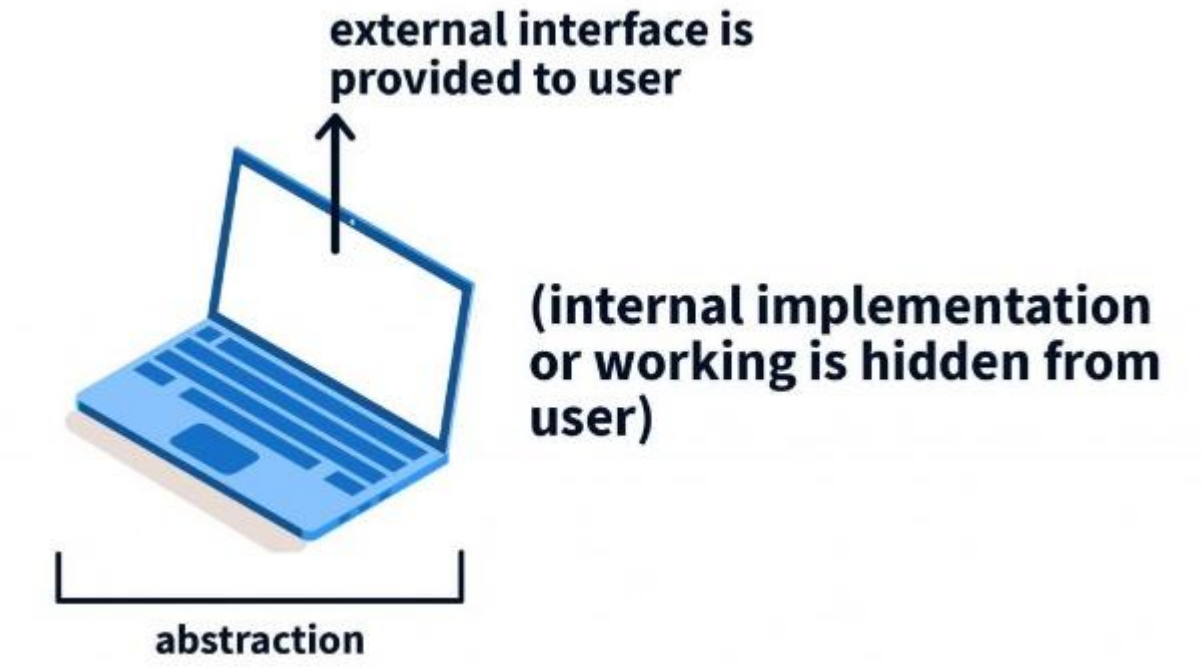


Abstraction

Abstraction involves showing only necessary details to the user and hiding irrelevant details or unnecessary details from the user. It is an extended form of Encapsulation and also serves a security purpose. As complex code is hidden from the user so it becomes easier for the user to use that software and also that software is easier to maintain.

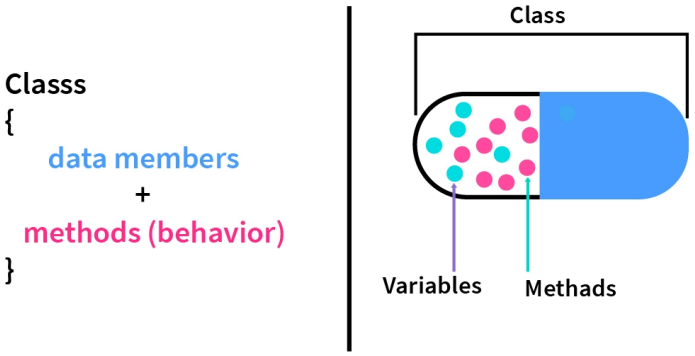
You can take a real-life example of a laptop or computer which you are using. On a laptop you can see the screen, use the keyboard and type something, you can use the mouse pad and can do all your work on the laptop. But do you know its internal details, how that keys are working and how the motherboard of the laptop is working, and how the picture is showing up on the screen? So here, all the internal

implementation of the laptop is hidden from the user and it is separated from its external interface. You can use the laptop without knowing its internal implementation. So abstraction provides simple and easy-to-use interfaces for the user.



What is Encapsulation?

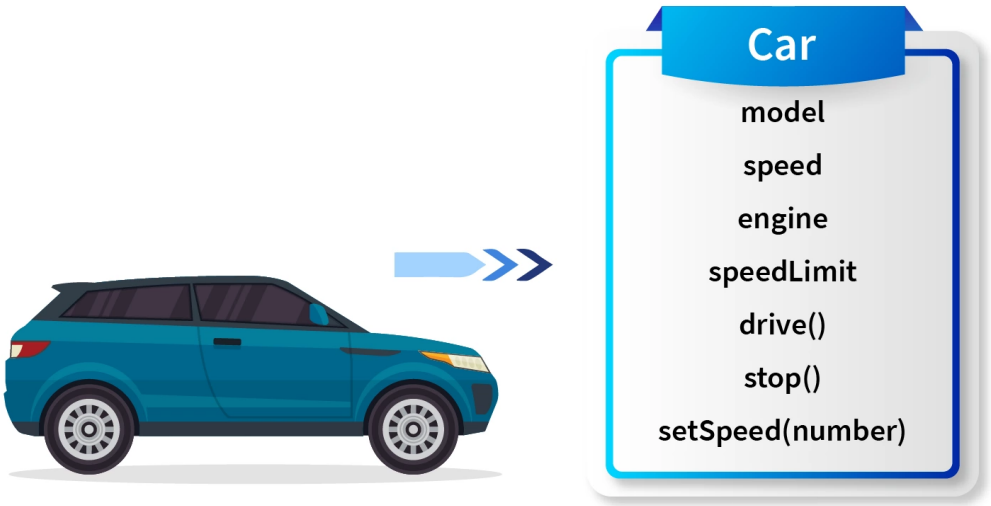
Encapsulation is a technique to bind all the data in a single unit along with methodds or functions that operate on that data. With the help of encapsulation we bind all the internal workings together. Its just like a capsule in which we can store multiple things and consume them.



Let's take an example of a car. In a car we have so many functionalities like drive(), stop(), setSpeed(number) etc. which helps us to control the car while driving, apart from this we have various properties of a car like model, speed, engine, speedLimit etc. which gives us the various information about the car. All these functions and properties are bound in a single unit called a car.

Similarly, in code, we wrap all these functions and properties. Functions are known as member functions and properties are known as member variables.

Encapsulation



See in the above figure we have a class named as car and we have all the member functions and member variables wrapped inside this single unit and this binding of data is known as encapsulation.

Encapsulation also helps us to hide the data and gives power to the programmer to define the access of data. Some data is only available to use in the class, some data is accessible by the public. That's why encapsulation is very important to make things easy.

Advantages of Encapsulation

1. It makes code modular and binds all properties and behaviours of the same entity in a single unit.
2. Because all similar things are wrapped in a single unit it becomes easy to do maintenance.
3. Code becomes more modular because of encapsulation.
4. Coders can specify the access of data and decide who can access it.

5. We can validate the data before assigning it to the variable. Fuel in the car can never be negative.

What is Abstraction?

Abstraction is a way to hide complexities and give a simple user interface to the user. With the help of abstraction, we hide internal complexities and make the system more user-friendly.

In a car, we have so many things to understand the abstraction, when we use brakes we don't know how it works, when we rotate the steering wheel we don't know how it turns the car. In abstraction, we just know what things do rather than how they do. All the complexities are hidden and only a simple user interface is provided to work.

When we use the accelerator of the car there are so many complex things happening under the hood, there is an engine that has a piston that takes fuel to generate energy with the laws of thermodynamics and this energy is converted into rotational energy to move it forward. As we go more specific about details things start to become more complex, but as we start to think about the accelerator only as an interface to move the car forward it is more abstract and easy to use.

Why do we Need abstraction?

Let's imagine the scenario where we are using brakes to stop or reduce the speed of a car. If we require to know how it works and if we are exposed to its internal working it becomes very difficult to use it. If this complex system of breaks is exposed to someone else he can make changes in it and perform malpractices. So to overcome this issue abstraction is used, abstraction provides both securities as well as it gives a simple user interface by hiding internal complexities.
In coding, consider the following image: only a few functions are accessible and others are hidden from the end-user of the program.

This way we can hide complexities and internal working of code and hide it from others to secure core functionalities of the program.

Advantages of Abstraction

1. Abstraction helps to hide complexities from the user to give a simple user interface.
2. It improves security just by showing essential details.
3. Due to abstraction, the user only focuses on what the object does instead of how it does.
4. Because of abstraction we can update the internal implementation by keeping the same interface so users will not have to adjust. For example, disk brakes and drum brakes have different internal working but both of them are accessed with a similar interface.

What is "inheritance" in Java?

Inheritance is a core principle in object-oriented programming. It allows an object to inherit the properties of a different class. The class the properties are inherited from is known as the superclass, and the one that inherits the properties is the subclass.

Inheritance

As the name suggests, inheritance is a mechanism in which one class can inherit the property of another class. In other words, the child class can inherit the property of the parent class. It’s just like a child-parent relationship in real life where a son or daughter is inherited by the properties of their parents.

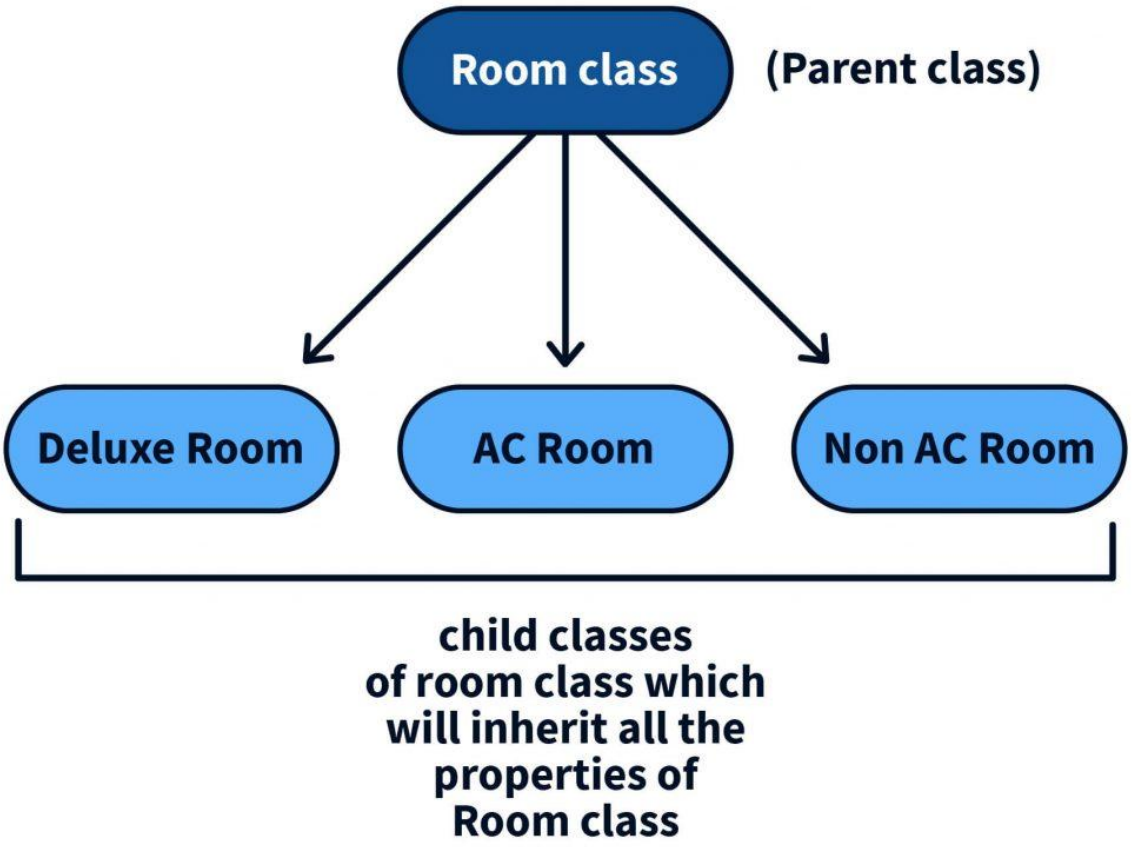
Inheritance promotes the reusability of code. If you want to create a new class and that class has some fields or code common to another class, you can use that class as the parent class and can inherit that class. So in this way, you don’t have to write that same piece of code

again and can use that code in the child class or class which you are creating. Hence that code is reused.

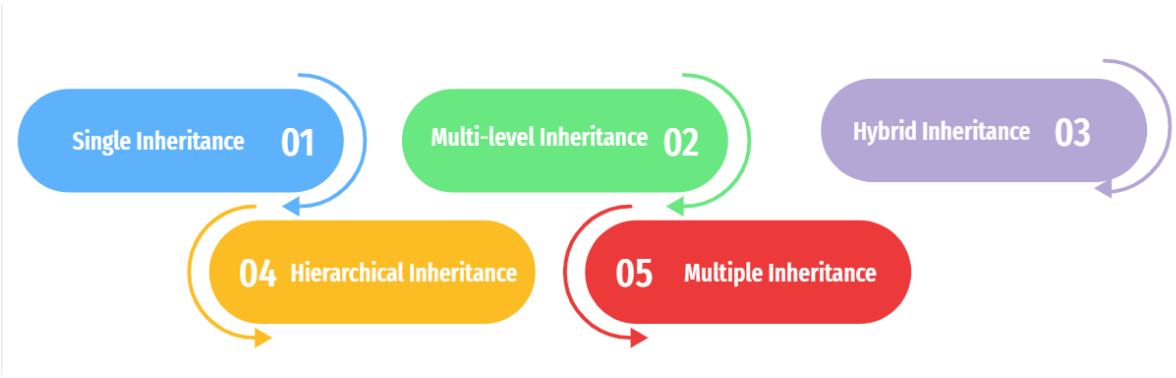
The syntax for inheritance is “extends” keyword is used to inherit another class.

```
class child_class extends parent_class {
  //fields and methods
}
```

You can take the example of that “Hotel Management system” which we have discussed in the above examples also. In that system, Rooms in the hotel can be of many types: deluxe rooms, AC rooms, Non AC rooms, Single room, Double room, etc. All rooms are subtypes of class Room or have some fields common to the class “Room” like room_rent, room_status, etc, it’s just like they can have some extra fields or methods. So here we can create child classes for parent class “Room” and all that child classes will inherit that “Room” class. This is the concept of inheritance.



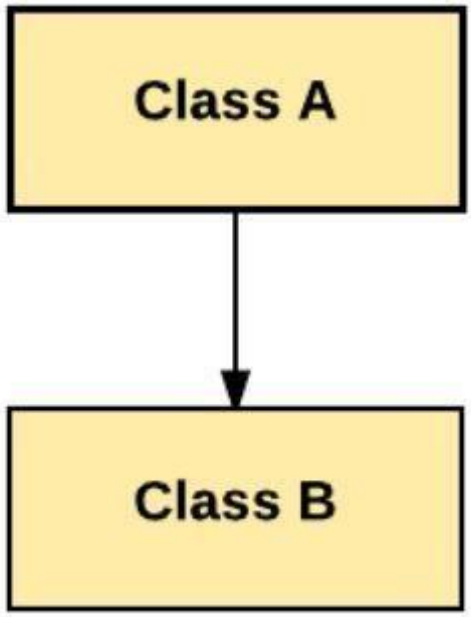
Types of Java Inheritance



Single level inheritance

As the name suggests, this type of inheritance occurs for only a single class. Only one class is derived from the parent class. In this type of inheritance, the properties are derived from a single parent class and

not more than that. As the properties are derived from only a single base class the reusability of a code is facilitated along with the addition of new features. The flow diagram of a single inheritance is shown below:



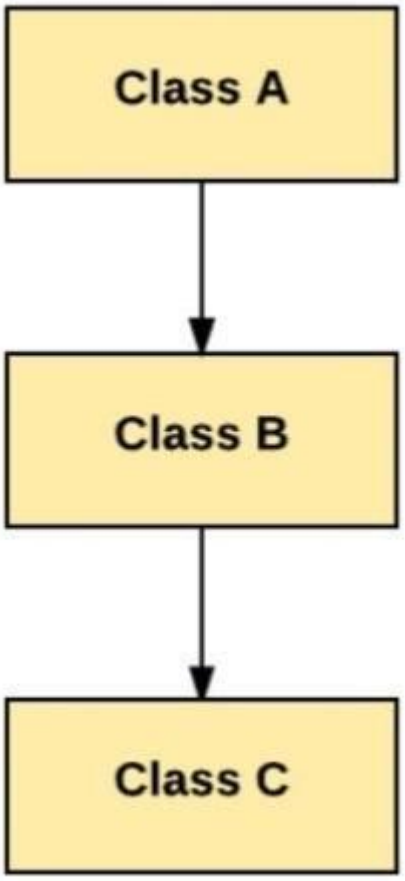
Two classes Class A and Class B are shown in Figure 2, where Class B inherits the properties of Class A.
Example: An example of a code applying single-level inheritance

```
Class A
{
   public void methodA()
   {
     System.out.println("Base class method");
   }
}

Class B extends A
{
   public void methodB()
   {
     System.out.println("Child class method");
   }
   public static void main(String args[])
   {
     B obj = new B();
     obj.methodA(); //calling super class method
     obj.methodB(); //calling local method
  }
}
```

Multi-level Inheritance

The multi-level inheritance includes the involvement of at least two or more than two classes. One class inherits the features from a parent class and the newly created sub-class becomes the base class for another new class.

As the name suggests, in the multi-level inheritance the involvement of multiple base classes is there. In the multilevel inheritance in java, the inherited features are also from the multiple base classes as the newly derived class from the parent class becomes the base class for another newly derived class.



From the flow diagram, we can observe that Class B is a derived class from Class A, and Class C is further derived from Class B. Therefore the concept of grandparent class comes into existence in multi-level inheritance. However, the members of a grandparent's class cannot be directly accessed in Java.

Therefore, Figure shows that Class C is inheriting the methods and properties of both Class A and Class B.

An example of multi-level inheritance is shown below with three classes X, Y, and Z. The class Y is derived from class X which further creates class Z.

Example: An example of multi-level inheritance

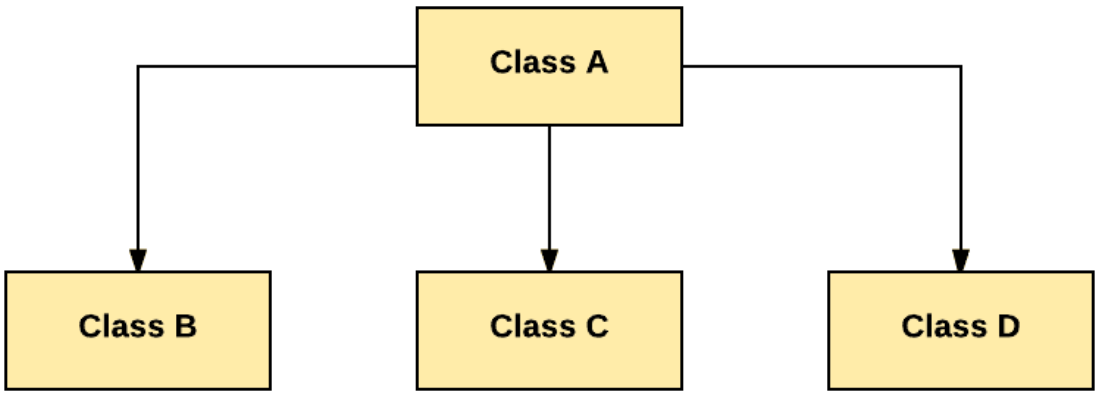
```
Class X
{
   public void methodX()
   {
     System.out.println("Class X method");
   }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
   public void methodZ()
   {
     System.out.println("class Z method");
   }
   public static void main(String args[])
   {
     Z obj = new Z();
     obj.methodX(); //calling grand parent class method
     obj.methodY(); //calling parent class method
     obj.methodZ(); //calling local method
  }
}
```

Hierarchical Inheritance

The type of inheritance where many subclasses inherit from one single class is known as Hierarchical Inheritance.

Hierarchical Inheritance a combination of more than one type of inheritance.

It is different from the multilevel inheritance, as the multiple classes are being derived from one superclass. These newly derived classes inherit the features, methods, etc, from this one superclass. This process facilitates the reusability of a code and dynamic polymorphism (method overriding).



In Figure, we can observe that the three classes Class B, Class C, and Class D are inherited from the single Class A. All the child classes have the same parent class in hierarchical inheritance.

Example: An example of code showing the concept of hierarchical inheritance

```
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
}}
```

The above code produces the output

```
meowing…
eating…
```

Other than these types of inheritance in Java, there are other types known as multiple inheritances and hybrid inheritance. Both types are not supported through classes and can be achieved only through the use of interfaces.

Multiple Inheritance

Multiple inheritances is a type of inheritance where a subclass can inherit features from more than one parent class. Multiple inheritances should not be confused with multi-level inheritance, in multiple inheritances the newly derived class can have more than one superclass. And this newly derived class can inherit the features from these superclasses it has inherited from, so there are no restrictions. In java, multiple inheritance can be achieved through interfaces.

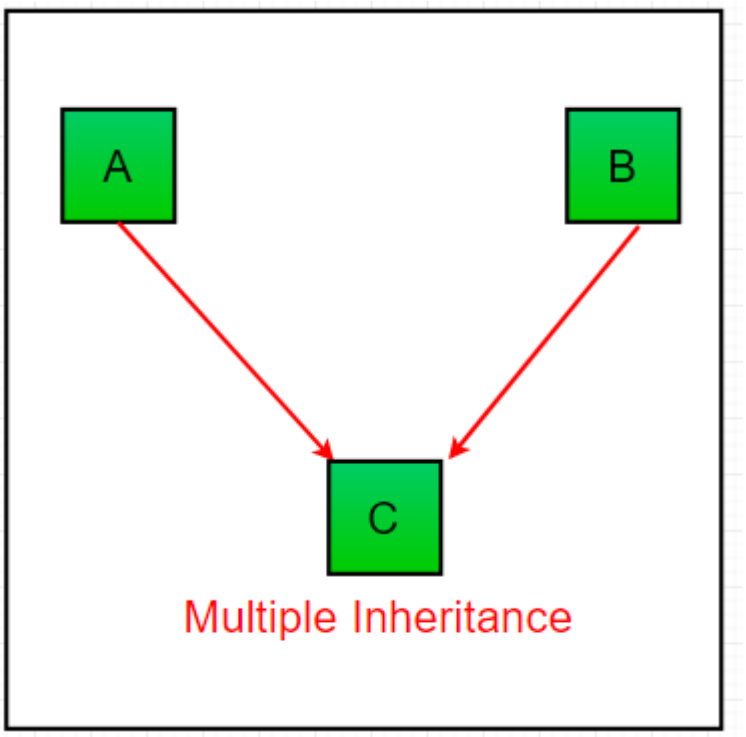
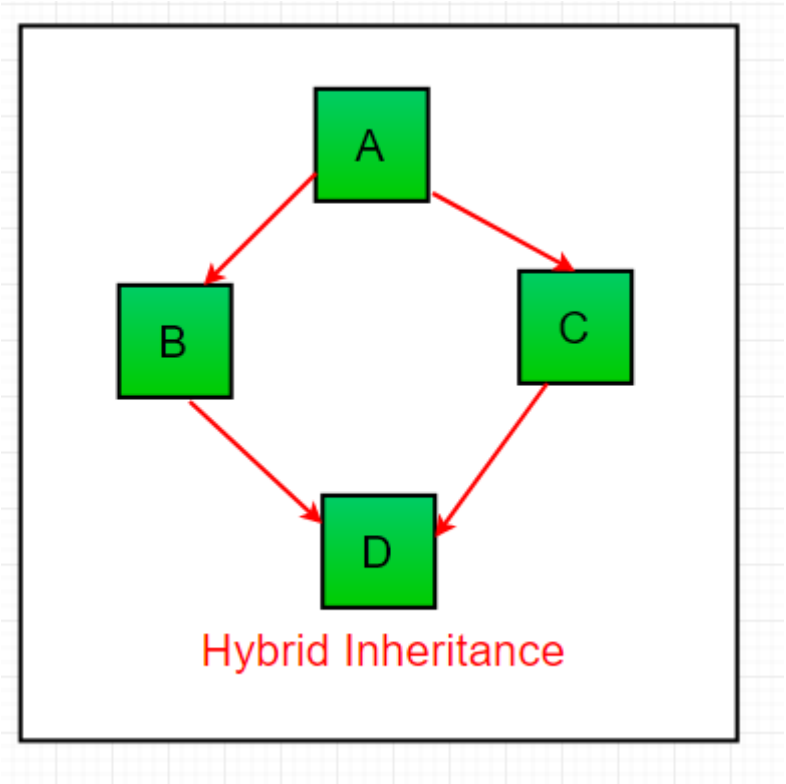


Figure shows that Class C is derived from the two classes Class A and Class B. In other words it can be described that subclass C inherits properties from both Class A and B.

Hybrid Inheritance

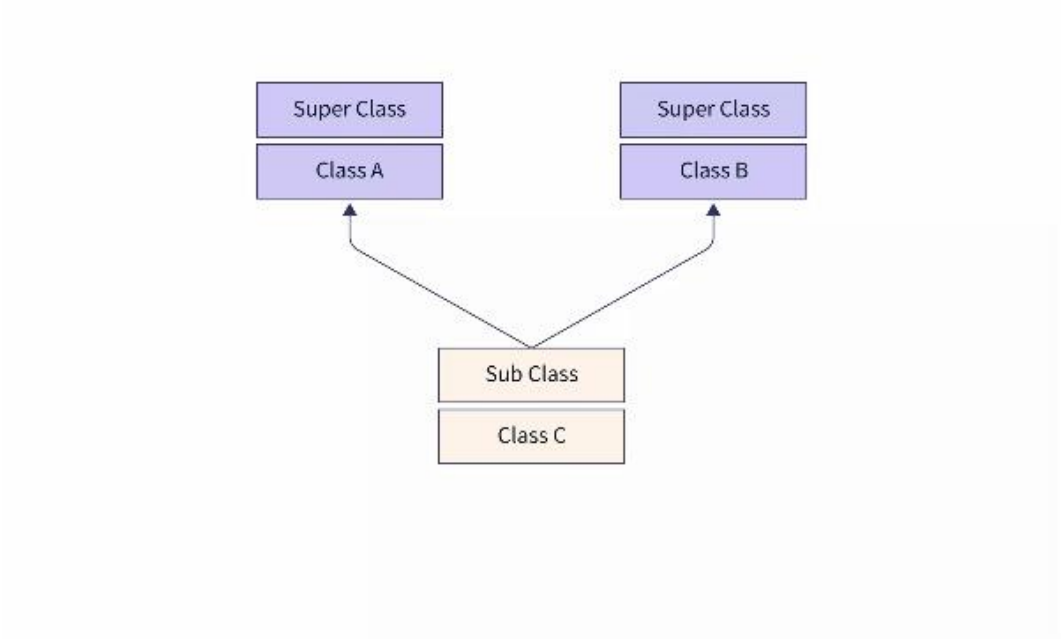
Hybrid inheritance is a combination of more than two types of inheritances single and multiple. It can be achieved through interfaces only as multiple inheritance is not supported by Java. It is basically the combination of simple, multiple, hierarchical inheritances.



Does Java Support Multiple Inheritance?

Inheritance is a fundamental concept of Object-Oriented Program(OOP). It is one of the strongest pillars of OOP. By definition, Inheritance is the process in which a class inherits all the properties (including methods, functions, variables) of another class. However, Multiple Inheritance is the process in which a class inherits properties from more than one class. Java doesn't support Multiple Inheritance. However, Multiple Inheritance in Java can be achieved using Interfaces.

- Multiple Inheritance in Java is the process in which a subclass inherits more than one Superclass.
- In the below image, we can observe that Class C(sub-class) inherits from more than one superclass i.e. Class A, Class B. This is the actual concept of Multiple Inheritance



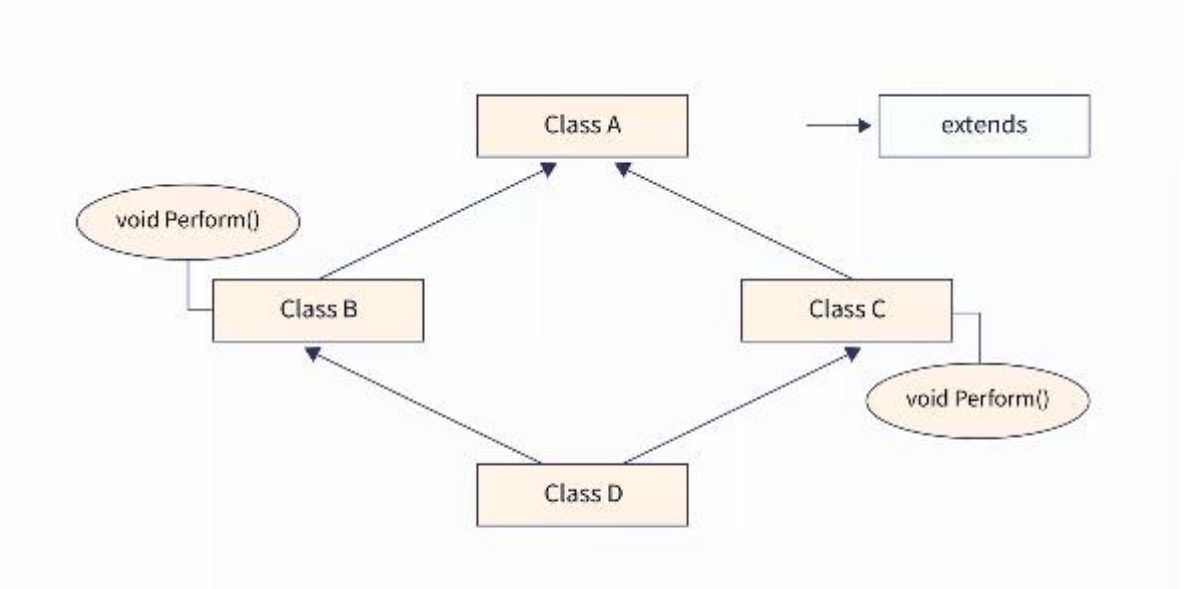
- Many real-world examples of Multiple Inheritance exist, For example, Consider a newly born baby, inheriting eyes from mother, nose from father. Got it right!!
- Observe the below program for Multiple Inheritance Program in Java-
    - The below Java program has three classes A, B, C
    - Class A, class B contains a method with the same name i.e. execute()
    - Class C inherits both class A, class B
    - In the Main class we created an object of class C with the name obj itself.

```
class A
{
   public void execute()
   {
      System.out.println("Hi.. Executing From Class A");
   }
}
class B
{
   public void execute()
```

```
    {
        System.out.println("Hi.. Executing From Class B");
    }
}
class C extends A, B
{

}
public class Main
{
        public static void main(String[] args)
        {
        C obj = new C();
        obj.execute();
        }
}
```

- On executing the above code, we could find an error i.e. an arrow mark pointing towards class C.
- This is because the compiler is confused about whether to call the execute()from class A or class B.
- Implementation of Multiple Inheritance in Java is not supported by default to avoid several ambiguity issues.
- One kind of ambiguity is the Diamond Problem, Lets understand this ambiguity with the help of the below image-
    - From the below image, we could observe that class A is inherited by class B, C.
    - Also, observe that class B, C is inherited by class D. This is where the actual ambiguity arises.
    - Assume that classes B, C both contain the same method with the same signature, if an object is instantiated for class D, and this object is used to call the method which is present in both class B, C. Take a moment to understand the scenario.

- Compiler would be confused because compiler does not know which class it should call to execute the perform() as it is present in both the classes.
- This is the actual reason why Multiple Inheritance in Java is not supported.



- Let's see how this could be managed by using the interface.
- Observe the below code
  - In the below code, we have two Interfaces i.e. Interface A, Interface B along with an Implementation class C and Main class.
  - Here, Interface A, Interface B contain the same method i.e. execute().
  - The above two Interfaces are implemented by the Implementation Class C.

```
interface A
{
  public void execute(int num1);
}
interface B
{
  public void execute(int num1);
```

```
}
class C implements A, B
{
  public void execute(int num1)
  {
    System.out.println("Hello.. From Implementation Class!!");
  }
}
public class Main
{
  public static void main(String[] args)
  {
     C obj = new C();
     obj.execute(16);
  }
}
```

Output

```
Hello.. From Implementation Class!!
```

From the above output, we could observe that though Interface A, Interface B contain the same method with the same signature, its implementation is provided in the Implementation Class only as interface methods are abstract by default so they don't contain definition due to which there is only one definition of execute() and object of child class had to only call that.

- By now, you would have a clear idea of how we could get rid of the Diamond problem by Implementing Multiple Inheritance in Java through Interface.

Does Java Support Goto Statements?

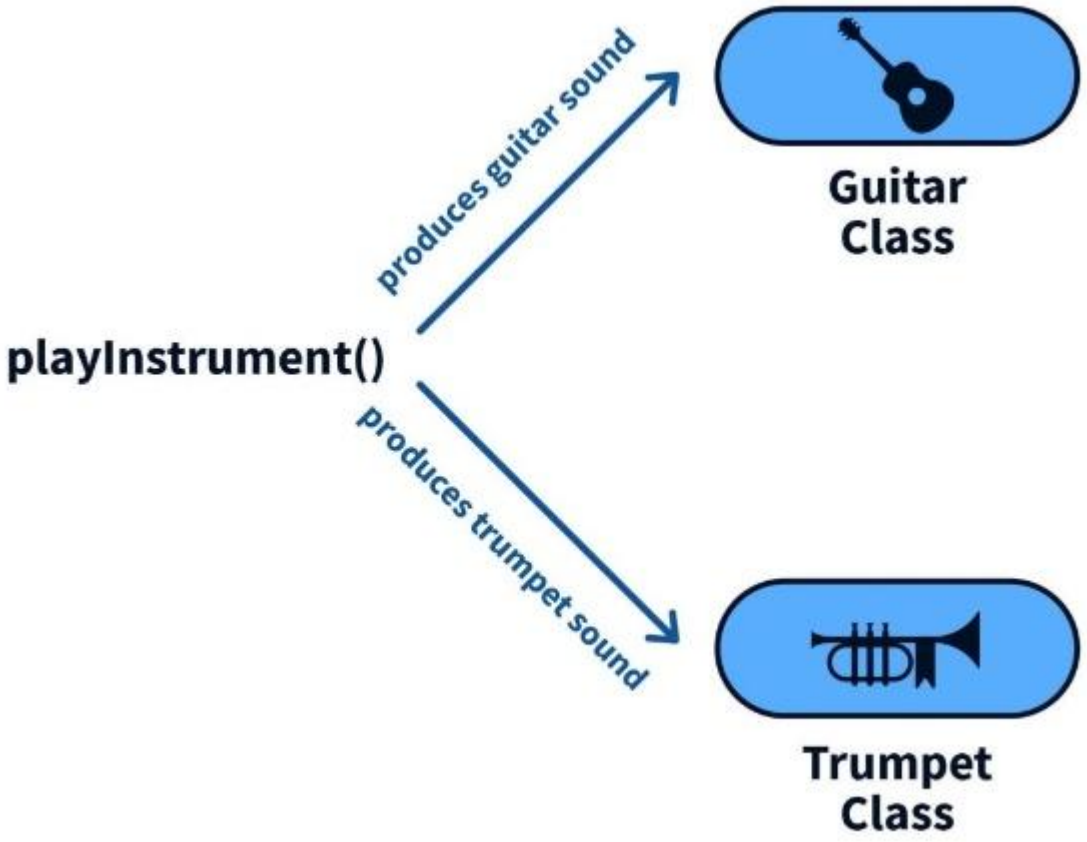
Java does not support goto statements while C and C++ supports goto statements. The Goto statement is a jumping statement. If you want to make a simple program then you should not use a goto statement, use of goto statement in any program makes the program very complex.

So a programmer should try to ignore the use of goto statement, Java focus to create simple code. That's why Java does not support goto statements.

Polymorphism

Polymorphism means one thing has different forms. Polymorphism allows a member function of a class to behave differently based on the object that will call it. Polymorphism occurs when classes are related through inheritance.

Suppose there is a function playInstrument() in parent class “Instrument” and Instrument class have two child classes or sub-classes: “Guitar“ and “Trumpet” and playInstrument will return the sound produced by the instrument which will call that function. So if the Guitar class object will call playInstrument then Guitar sound will be the output and if the Trumpet class object will call playInstrument then Trumpet sound will be the output.



Polymorphism can be achieved through function overloading and function overriding.

- **Function overriding**: In function overriding child class can exhibit different implementations of the same function which has been defined in parent class also, according to its usages.

The above example of “Instrument” class is an example of Function overriding.

- **Function overloading**: Polymorphism is also achieved through function overloading. In function overloading, methods can have the same name but the number of arguments is different in function calls. Results will be different according to the number of arguments with which the function is called.

You can take a real-life example. Suppose you created a function to calculate areas of a figure to calculate the area of rectangle and square. As you know, the rectangle area is length breadth and square’s area is side side so the rectangle area function should take 2 arguments length and breadth whereas square area will take one argument that is side.

So instead of creating functions with separate names, you can create two functions with the same name but the number of arguments is different for both functions.

```
//1- for square
int Findarea(int side) {
  return side * side;
}
//for rectangle
int Findarea(int length, int breadth) {
  return length * breadth;
}
```

findArea(3) call will go to 1 function as the number of arguments in function is one and findArea(2,5) call will go to 2 function as the

number of arguments in the function call is two. This is the concept of function overloading.

Example

```
// Parent class to illustrate run-time polymorphism
class Parent {
  // creating print method
  void print() {
    System.out.println("Hi I am parent");
  }
}
// Child class extends Parent class
class Child extends Parent {
  // overriding print method
  void print() {
    System.out.println("Hi I am children");
  }
}
// Overload class to illustrate compile-time polymorphism
class Overload {
  // Creating a statement method
  void statement(String name) {
    System.out.println("Hi myself " + name);
  }
  // overloading statement method
  void statement(String fname, String lname) {
    System.out.println("Hi myself " + fname + " " + lname);
  }
}
public class Main {
  public static void main(String[] args) {
    // creating instance of parent
    Parent obj1;
    obj1 = new Parent();
    obj1.print();
    obj1 = new Child();
```

```
    obj1.print();

    // creating instance of overload
    Overload obj2 = new Overload();
    obj2.statement("Soham.");
    obj2.statement("Soham", "Medewar.");
  }
}
```

Output:

```
Hi I am a parent
Hi I am children
Hi myself Soham.
Hi myself Soham Medewar.
```

In the above example, the Child class extends the Parent class and the print method is overridden, this represents run-time polymorphism in Java.

In the overload class, the statement function is overloaded, this represents compile-time polymorphism in Java.

Types of Polymorphism in Java

There are two main types of polymorphism in Java.

1. Compile-time polymorphism

This type of polymorphism in Java is also called static polymorphism or static method dispatch. It can be achieved by method overloading.

In other words, a class can have multiple methods of the same name and each method can be differentiated either by bypassing different types of parameters or bypassing the different number of parameters.

Example 1

Passing different numbers of arguments to the function.

```
class Compiletime {
  // perimeter method with single argument
  static int perimeter(int a) {
    return 4 * a;
  }
  // perimeter method with two arguments (overloading)
  static int perimeter(int l, int b) {
    return 2 * (l + b);
  }
}
class Polymorphism {
  public static void main(String[] args) {
    // calling perimeter method by passing single argument
      System.out.println("Side of square : 4\nPerimeter of square will be : " + Compiletime.perimeter(4) + "\n");
    // calling perimeter method by passing two arguments
      System.out.println("Sides of rectangle are : 10, 13\nPerimeter of rectangle will be : " + Compiletime.perimeter(10, 13));
  }
}
```

Output

```
Side of square : 4
Perimeter of square will be : 16

Sides of rectangle are : 10, 13
Perimeter of rectangle will be : 46
```

Explanation

In the above example, the “compile_time” class has two functions, both having the same name, but the first function has a single argument to pass, and another one has two arguments to pass.

Here, the perimeter function can calculate the area of squares and rectangles. In this way, two functions having the same name are distinguished, and compile-time polymorphism is achieved.

For the first time, when we call the "perimeter" method, we pass a single integer to the method, so the first method is evoked, and for the second time, we pass two integers to the method, and this time the second method is evoked.

Example 2

Passing different types of argument to the function.

```
class Compiletime{
   // contact method which takes two argument String and long
        static void contact(String fname, long number) {
                System.out.println("Name    :    "+fname+"\nNumber    :
"+number);
        }
    // contact method which takes two arguments and both are Strings
(overloading)
        static void contact(String fname, String mailid) {
                System.out.println("Name     :     "+fname+"\nEmail     :
"+mailid);
        }
}
class Polymorphism{
   public static void main(String[] args) {
       // calling first contact method
       Compiletime.contact("Soham", 1234567890);
       System.out.print("\n");
```

```
        // calling second contact method
        Compiletime.contact("Soham", "soham@mail.com");
    }
}
```

Output

```
Name : Soham
Number : 1234567890

Name : Soham
Email : soham@mail.com
```

Explanation

In the above example, the "compile_time" class has two functions, both having the same name, but in the first function, we pass a string and long as an argument, and in the second function, we pass two strings as an argument.

It shows that we can save the person's contact by mobile number or email. In this way, compile-time polymorphism is achieved.

For the first time, when we call the "contact" method, we pass a name and mobile number (string and long) as arguments, so the first contact method is evoked.

The second time while calling the "contact" method, we pass a name and email (string and string) as arguments, so the second contact method is evoked.

2. Runtime Polymorphism

Runtime polymorphism is also called Dynamic method dispatch. Instead of resolving the overridden method at compile-time, it is resolved at runtime.

Runtime polymorphism in Java occurs when we have two or more classes, and all are interrelated through inheritance. To achieve runtime polymorphism, we must build an "IS-A" relationship between classes and override a method.

Method overriding

If a child class has a method as its parent class, it is called method overriding.

If the derived class has a specific implementation of the method that has been declared in its parent class is known as method overriding.

Rules for overriding a method in Java

There must be inheritance between classes.

The method between the classes must be the same(name of the class, number, and type of arguments must be the same).

Example :

```
//parent class Animal
class Animal{
    // creating place method
        void place(){
                System.out.println("Animals live on earth.");
        }
}
// Dog class extends Animal class
class Dog extends Animal{
    // overriding place method
        void place(){
                System.out.println("Dog lives in kennel.");
        }
}
// Horse class extends Animal class
```

```
class Horse extends Animal{
    // overriding place method
        void place(){
                System.out.println("Horse lives in stable.");
        }
}
// Rabbit class extends Animal class
class Rabbit extends Animal{
    // overriding place method
        void place(){
                System.out.println("Rabbit lives in burrow.");
        }
}
class Polymorphism{
    public static void main(String[] args) {
        // creating object of Animal class
         Animal A = new Animal();
         A.place();
         A = new Dog();
         A.place();
         A = new Horse();
         A.place();
         A = new Rabbit();
         A.place();
    }
}
```

Output

```
Animals live on earth.
Dog lives in kennel.
Horse lives in stable.
Rabbit lives in burrow.
```

Explanation

In the above example, the "Animal" class is parent class, and "Dog", "Horse", "Rabbit" are its derived class where the "place" method is overridden.

We have created a instance of animal class. When an object of every child class is created, the method inside the child class is called. As, the parent class method is overridden by child class.

Example :

```
// parent class Bank
class Bank{
    // creating rateOfInterest method
        float rateOfInterest(){
                return 0;
        }
}
// ICICI class extends Bank class
class ICICI extends Bank{
    // overriding rateOfInterest method
        float rateOfInterest(){
                return 5.5f;
        }
}
// SBI class extends Bank class
class SBI extends Bank{
    // overriding rateOfInterest method
        float rateOfInterest(){
                return 10.6f;
        }
}
// HDFC class extends Bank class
class HDFC extends Bank{
    // overriding rateOfInterest method
        float rateOfInterest(){
                return 9.4f;
        }
```

```
}
class Polymorphism{
    public static void main(String[] args) {
        // creating variable of Bank class
         Bank B;
         B = new ICICI();
         System.out.println("Rate of interest of ICICI is: "+B.rateOfInterest());
         B = new SBI();
         System.out.println("Rate of interest of SBI is: "+B.rateOfInterest());
         B = new HDFC();
         System.out.println("Rate of interest of HDFC is: "+B.rateOfInterest());
    }
}
```

Output

```
Rate of interest of ICICI is: 5.5
Rate of interest of SBI is: 10.6
Rate of interest of HDFC is: 9.4
```

Explanation

In the above example, the “bank” class is parent class and “ICICI”, “SBI”, and “HDFC” are its derived class where the “rate()” method is overridden.

Implementing abstraction in Java

Abstraction in Java can be achieved in two ways:

1. Using abstract classesAbstract classes achieve partial abstraction as concrete methods can also be defined in them.

2. Using interfacesInterfaces achieve complete abstraction as only abstract methods can be defined in them.

Abstraction using Abstract Class

Abstract class

It is a class that cannot be instantiated. That is, you cannot create objects of an abstract class using the new keyword.

Abstract method

It is a method that has no code block.

You can implement abstraction in Java by declaring a class as abstract using the abstract keyword. Abstract methods are generally declared in abstract classes. But implemented methods can also be written in them. It is not compulsory to include abstract methods in abstract classes. Abstract methods (unimplemented methods) are declared using the abstract keyword.

General Syntax:

```
public class abstract class_name {
    public abstract returnType method_name();
}
public class subclass_name extends abstract_class {
    public returnType method_name() { //method implementation
    }
}
```

When should you use abstract classes?

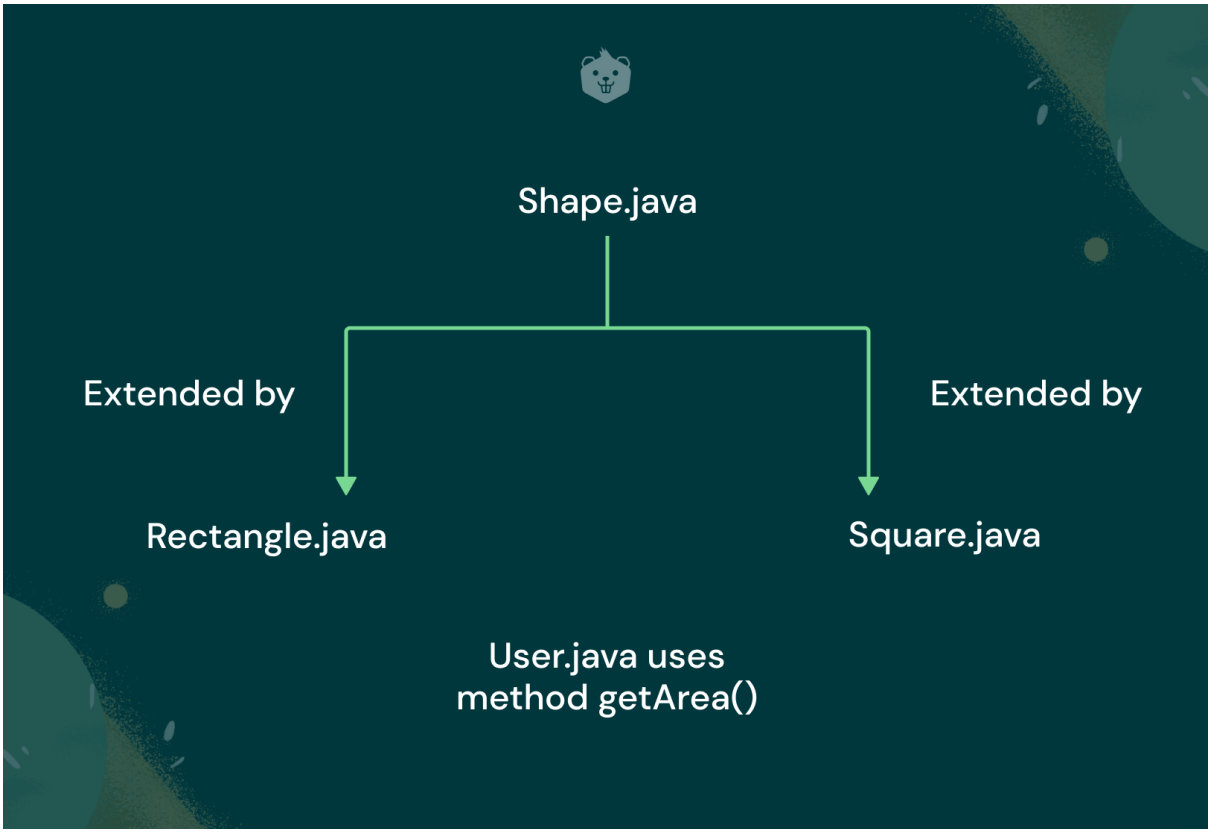
Abstract classes should mainly be used when:

1. You need different implementations of attribute(s) or method(s) across different subclasses that implement these attribute(s) or method(s).
2. You have a design in mind, but you don't know how to (or maybe you cannot) implement it yet.

Examples of abstraction using abstract class

Usage 1

Here, an abstract class Shape has an abstract method getArea(). The subclasses Rectangle and Square inherit the properties of Shape, override and implement the method getArea().Thereby, a user can simply call the method getArea() on the objects of Rectangle and Square and get their area.




```
package pack;
```

```
public abstract class Shape { //abstract method
    getArea() public abstract double getArea();
}
```

```
import pack.Shape;
public class Rectangle extends Shape {
    double length;
    double width;
    public Rectangle(double l, double w) {
        length = l;
        width = w;
    }
        @Override //getArea() is overridden and implemented by
Rectangle
    public double getArea() {
        return length * width;
    }
}
```

```
import pack.Shape;
public class Square extends Shape {
    double length;
    public Square(double l) {
        length = l;
    }
        @Override //getArea() is overridden and implemented by
Rectangle
    public double getArea() {
        return length * length;
    }
}
```

```
import pack.*;
public class User {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(4, 5);
        System.out.println(rect.getArea());
        Square sq = new Square(3);
        System.out.println(sq.getArea());
    }
}
```

Usage 2

Think that you are designing the Instagram app, and you have an idea for a feature -- Reels. (Reels are 15-second videos that users can upload to their profile)

Reels have the potential to become popular and to be used a lot by users.

But, your company, Instagram, currently doesn't have the storage space required to be able to host millions of videos on its servers. But you need to launch Instagram to the market very soon, as the timing is perfect for it.

When the time comes, and when you have enough storage, you want to launch the Reels feature.

Java can help you here by making the method that should implement Reels as abstract. Using inheritance, you can extend the class that has the abstract Reels method, and override it and implement it in the subclass.

Thereby, you have ensured that the Reels upgrade to Instagram occurs smoothly.

```
package pack;
public abstract class Instagram {
    public String logoColor = "pink";
    public String logoFont = "Roboto";
    public void photoPost() {
        System.out.println("Feature enabling users to post pictures");
    }
    public void chat() {
        System.out.println("Feature enabling users to chat");
    }
    public void storyPost() {
        System.out.println("Feature enabling users to post stories");
    }
    public abstract void Reels();
}
```

```
import pack.Instagram;
public class featureReels extends Instagram {
    public void Reels() {
        System.out.println("Feature enabling users to post reels");
    }
    public static void main(String args[]) {
        featureReels reel = new featureReels();
        reel.Reels();
    }
}
```

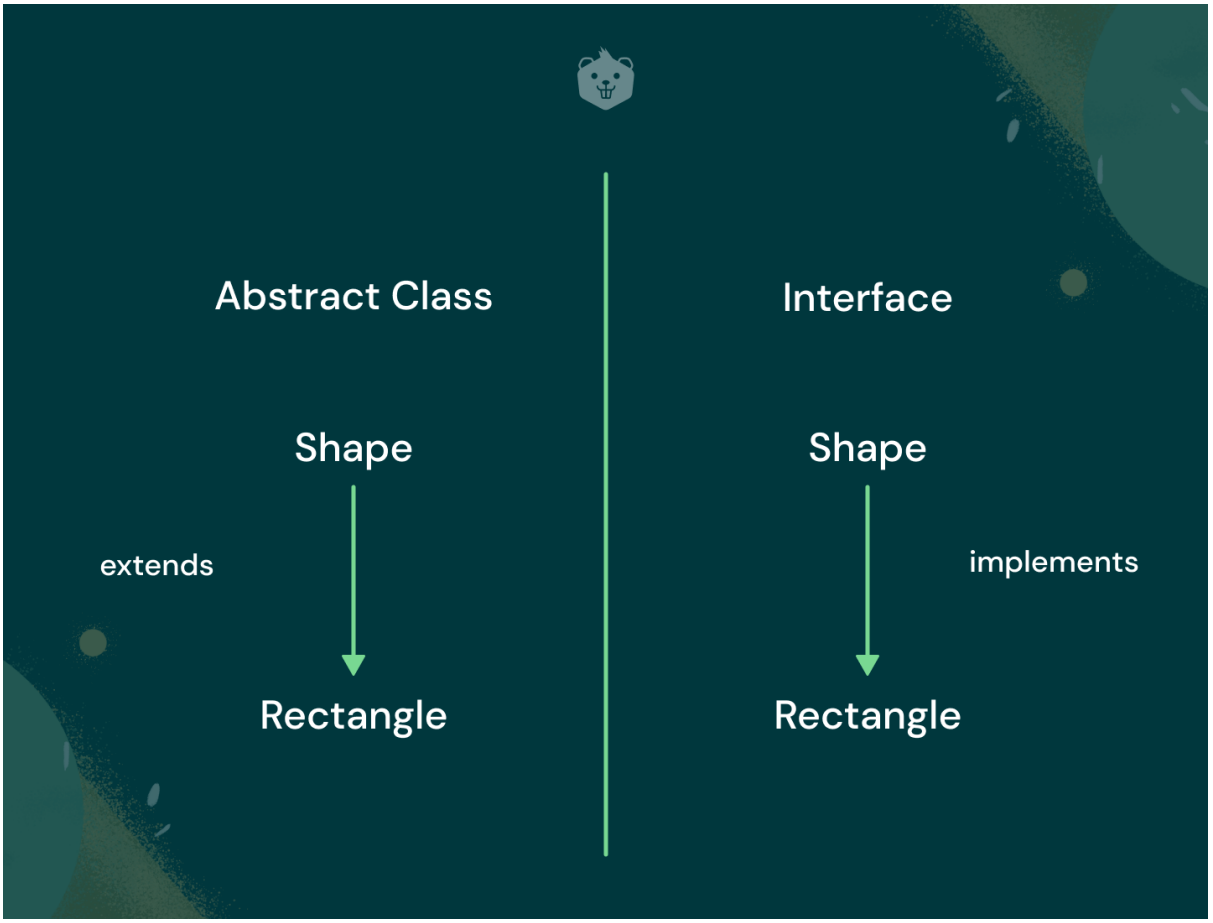
Output:

```
Feature enabling users to post reels
```

6 rules to follow when implementing Abstraction using abstract classes and methods

Abstraction using Interfaces

An interface can only have abstract methods and final attributes. All the abstract methods of an interface must be implemented unless the class implementing the interface is an abstract class.Methods that are declared in an interface are abstract by default.



General Syntax:

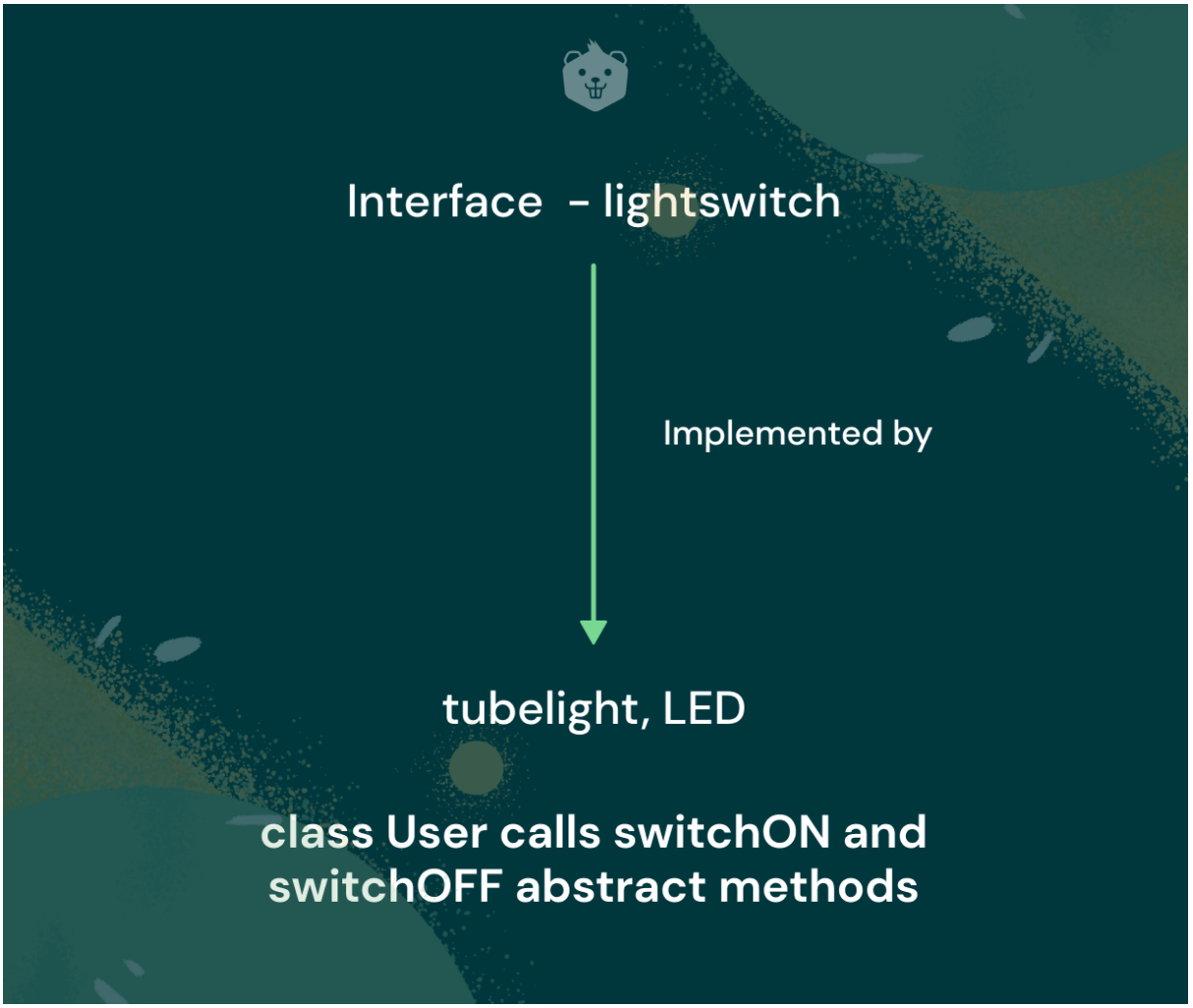
```
public interface interface_name {
    public abstract returnType method_name(); //or
    public returnType method_name();
}
class class_name implements interface_name {
    public returnType method_name() { //method implementation
}
```

When should you use interfaces?

Interfaces are used especially when multiple inheritance is needed. Multiple inheritance occurs when a subclass inherits the properties of more than one superclass.

Examples of abstraction using interface

Here, an interface light



Switch has two methods switch ON and switchOFF, which are implemented by a class lightBulb. A class User makes use of these methods, without knowing how the light gets turned on or off.

```
package pack;
public interface lightSwitch {
    public Boolean switchON();
    public Boolean switchOFF();
}
```

```
import pack.*;
public class tubeLight implements lightSwitch {
    public Boolean light;
```

```
    public Boolean switchON() {
        light = true;
        return light;
    }
    public Boolean switchOFF() {
        light = false;
        return light;
    }
}
```

```
import pack.*;
public class LED implements lightSwitch {
    public Boolean light;
    public Boolean switchON() {
        light = true;
        return light;
    }
    public Boolean switchOFF() {
        light = false;
        return light;
    }
}
```

```
import pack.*;
public class User {
    public static void main(String args[]) {
        lightSwitch bulb1 = new tubeLight();
        lightSwitch bulb2 = new LED();
        System.out.println(bulb1.switchON());
        System.out.println(bulb2.switchOFF());
    }
}
```

7 rules to follow when using interfaces

This and Super Keyword

The super keyword is used to represent an instance of the parent class and can invoke constructor of the parent class during inheritance, whereas the this keyword is used to represent an instance of the current class, also this keyword is used to refer the static members as well as invoke constructor of the current class.

this: this is the reserved keyword in Java that is used to invoke constructor, methods, static members of the current class.

super: super is the reserved keyword in Java that is used to invoke constructor, methods of the parent class. This is possible only when one class inherits another class(child class extends parent class).

Uses of “this” keyword in Java

- It can be passed as an argument in the method call.
- It can be used to return the current class instances.

- It refers to an instance variable and static variable of the current class.
- It can be passed as an argument in the constructor call.
- It is used to initiate a constructor of the current class.

Example 1

The “this” keyword is referring to an instance and a static variable in the same class. By this we can change the values of the variables that are declared inside the class. Also we can differentiate between the variables that are declared inside the classes and methods.

```
// Illustration class
class Illustration{
	// declaring an instance variable
	int instanceVar = 5;

	// declaring an static variable
	static int staticVar = 10;

	void Scaler(){
		int instanceVar = 20;
		// referring to the current class instance variable
		this.instanceVar = 50;

		int staticVar = 40;
		// referring to the current class static variable
		this.staticVar = 100;

		// printing the current class instance and static variable.
		System.out.println("Value of instance variable : "+this.instanceVar);
		System.out.println("Value of static variable : "+this.staticVar);
		System.out.println("Value of variables declared inside method : "+instanceVar+" "+staticVar);
	}
```

```
}
public class main {
    public static void main(String[] args) {
        // creating an instance of Illustration class
        Illustration obj = new Illustration();
        obj.Scaler();
    }
}
```

Output

```
Value of instance variable : 50
Value of static variable : 100
Value of variables declared inside method : 20 40
```

In the above example, the instanceVar is an instance variable, and the staticVar is a static variable inside the class. Both the variables are referred to in the "Scaler"method by using the "this" keyword. instanceVar is initialized with value 5, and staticVar is initialized with value 10. The scaler method reinitializes the instanceVar and staticVar using the "this" keyword with 50 and 100, respectively. There are other two variables having the same name as static variables. We are able to differentiate between variables inside the class and methods by using this keyword.

Example 2:

The “this” keyword is used to invoke current class method. Instead of calling all the methods of class differently inside the main class we can use this keyword, so that all the methods can be executed using a single call.

```
// Illustration class
class Illustration{
        // current class method
        void scaler(){
            System.out.print("My name is : ");
        }
```

```
        void name(){
       // invoking current class scaler method.
          this.scaler();
           System.out.println("Soham.");
        }
}
public class Main {
   public static void main(String[] args) {
        // creating an instance of Illustration class
        Illustration obj = new Illustration();
        obj.name();
   }
}
```

Output:

```
My name is : Soham.
```

In the above example, the Illustration class has two methods scaler and name. The scaler method is invoked inside the name method by using the “this” keyword. To call the scaler method, we don't need to write writing obj.scaler() instead we can call it inside name function using this keyword.

Example 3

The “this” keyword is used to invoke the constructor of the current class. Suppose a class has two constructor one is no argument constructor and another is parameterized constructor. So for invoking both the constructors by a single call is can be done using a this keyword.

```
// Illustration class
class Illustration{
        // simple constructor
        Illustration(){
                // invoking parameterized constructor
```

```
            this(10);
        }

        // parameterized constructor
        Illustration(int x){
            System.out.println("Current class parameterized
constructor invoked.");
            System.out.println("Number is : "+x);
        }
}
public class Main {
    public static void main(String[] args) {
        // creating an instance of Illustration class
        Illustration obj = new Illustration();
    }
}
```

Output

```
Current class parameterized constructor invoked.
Number is : 10
```

In the above example, there are two constructors. The no argument constructor is called while creating the object, and the parameterized constructor is called within the no argument constructor by using the “this” keyword. current class method.

Example 4

The “this” keyword is used to pass an argument in the method. At industry level it is used in event handlers or at places where we have to provide reference of one class to another class.

```
// Illustration class
class Illustration{
        // print method
        void print(Illustration ob){
                System.out.println("Method is invoked.");
        }
        void invoke(){
                // print method is invoked by passing this as an argument
                print(this);
        }
}
public class Main {
    public static void main(String[] args) {
        // creating an instance of Illustration class
        Illustration obj = new Illustration();
        obj.invoke();
    }
}
```

Output:

```
Method is invoked.
```

In the above example, we are passing the “this” keyword as an argument to call the print method in the invoke method.

Example 5

The “this” keyword is used to pass as an argument in the constructor call. By passing this as an argument in the constructor call we can

exchange data from one class to another. This makes exchange of data between multiple classes a lot easier.

```
// A class
class A{
	// instance variable
	B tmp;
	// parameterized constructor
	A(B tmp){
		this.tmp = tmp;
	}
	// print method
	void print(){
		System.out.println("The number is : "+tmp.val);
	}
}
// B class
class B{
	// instance variable
	int val = 50;
	// constructor
	B(){
		// creating instance of A
		// passing “this” as an argument in constructor call
		A obj = new A(this);
		obj.print();
	}
}
public class Main {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

Output

```
The number is : 50
```

In the above example, we have passed the "this" keyword as an argument in the constructor call. Here A and B are two classes; parameterized constructor of A is called in class B by creating an object of class A and passing "this" as a parameter.

Example 6

The “this” keyword is used to return the current class instance. Methods of the class can be called directly at the time of creating an object.

```
// illustration class
class Illustration{
        Illustration getIllustration(){
                // returing the instance of current class
                return this;
        }
        void print(){
                System.out.println("Hello World!");
        }
}
public class Main {
   public static void main(String[] args) {
        new Illustration().getIllustration().print();
   }
}
```

Output

```
Hello World!
```

In the above example, the getIllustration method returns the current class instance and through which we called the print method.

What is a “super” keyword in Java?

“super” is a special keyword in Java that is used to refer to the instance of the immediate parent class. Using the “super” keyword, we can refer to the immediate parent class's methods, constructor, instance variables, and many more. We cannot use the “super” keyword as an identifier in Java.

Uses of “super” Keyword in Java

- It is used to refer to an instance variable of an immediate parent class.
- It is used to invoke a method of an immediate parent class.
- It is used to invoke a constructor of an immediate parent class.

Example 1

The “super” keyword is used to refer to an instance variable of an immediate parent class.

```
// parent class
class Parent{
        int a = 50;
        String s = "Hello World!";
}
// child class extending parent class
class Child extends Parent{
        int a = 100;
        String s = "Happy Coding!";
        void print(){
                // referencing to the instance variable of parent class
                System.out.println("Number from parent class is :
```

```
"+super.a);
                System.out.println("String from parent class is :
"+super.s);

                // printing a and s of the current/child class
                System.out.println("Number from child class is : "+a);
                System.out.println("String from child class is : "+s);
        }
}
public class Main {
    public static void main(String[] args) {
        // creating instance of child class
        Child obj = new Child();
        obj.print();
    }
}
```

Output

```
Number from parent class is : 50
String from parent class is : Hello World!
Number from child class is : 100
String from child class is : Happy Coding!
```

In the above example, the child class extends the parent class. Both have two instance variables, a and s. In child class, the print method calls the instance variables of the parent class using the "super" keyword. In comparison, the instance variables of child class don't need one. It doesn't matter whether the instance variable of parent class and child class share the same name; they can hold different values and are not overridden.

Example 2

The “super” keyword is used to invoke an immediate parent class method.

```
// parent class
class Parent{
	// declaring display method parent class
	void display(){
		System.out.println("Hi i am parent method.");
	}
}
// child class extending parent class
class Child extends Parent{
	// declaring display method in Child class
	void display(){
		System.out.println("Hi i am child method.");
	}
	void print(){
		// invoking display method from parent class
		super.display();
		// display method from child class
		display();
	}
}
public class Main {
   public static void main(String[] args) {
	// creating instance of child class
	Child obj = new Child();
	obj.print();
   }
}
```

Output

```
Hi i am parent method.
Hi i am child method.
```

In the above example, child class extends parent class. Both have a display method. The print method in the child class invokes the display method of the parent class by using the “super” keyword. The example says that to invoke the methods in the child class from the parent class, we must use the “super” keyword.

Example 3

The “super” keyword is used to invoke an immediate parent class constructor.

```
// parent class
class Parent{
        // parent class constructor
        Parent(){
                System.out.println("Hi I am Parent class constructor.");
        }
}
// child class extending parent class
class Child extends Parent{
        // child class constructor
        Child(){
                // invoking parent class constructor
                super();
        }
}
public class Main {
    public static void main(String[] args) {
        // creating instance of child class
        Child obj = new Child();
    }
}
```

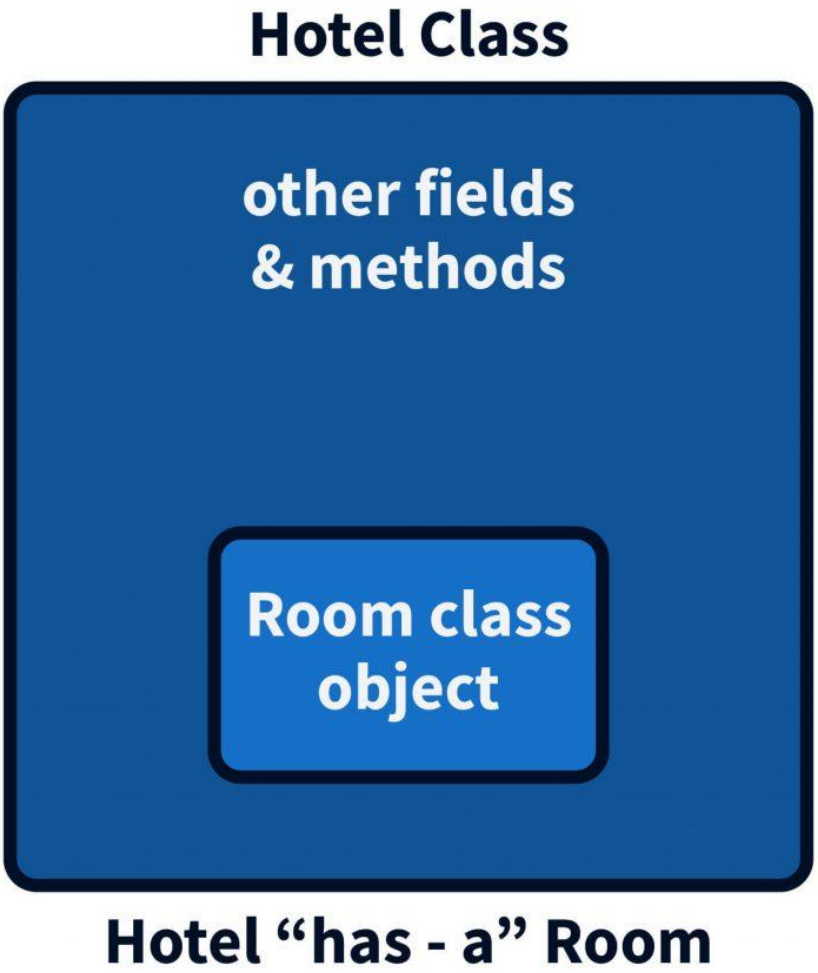
Output

```
Hi I am Parent class constructor.
```

In the above example, the child class extends parent class. While creating an object of the child class, the constructor of the child class is called, where the parent class constructor is invoked using the “super” keyword.

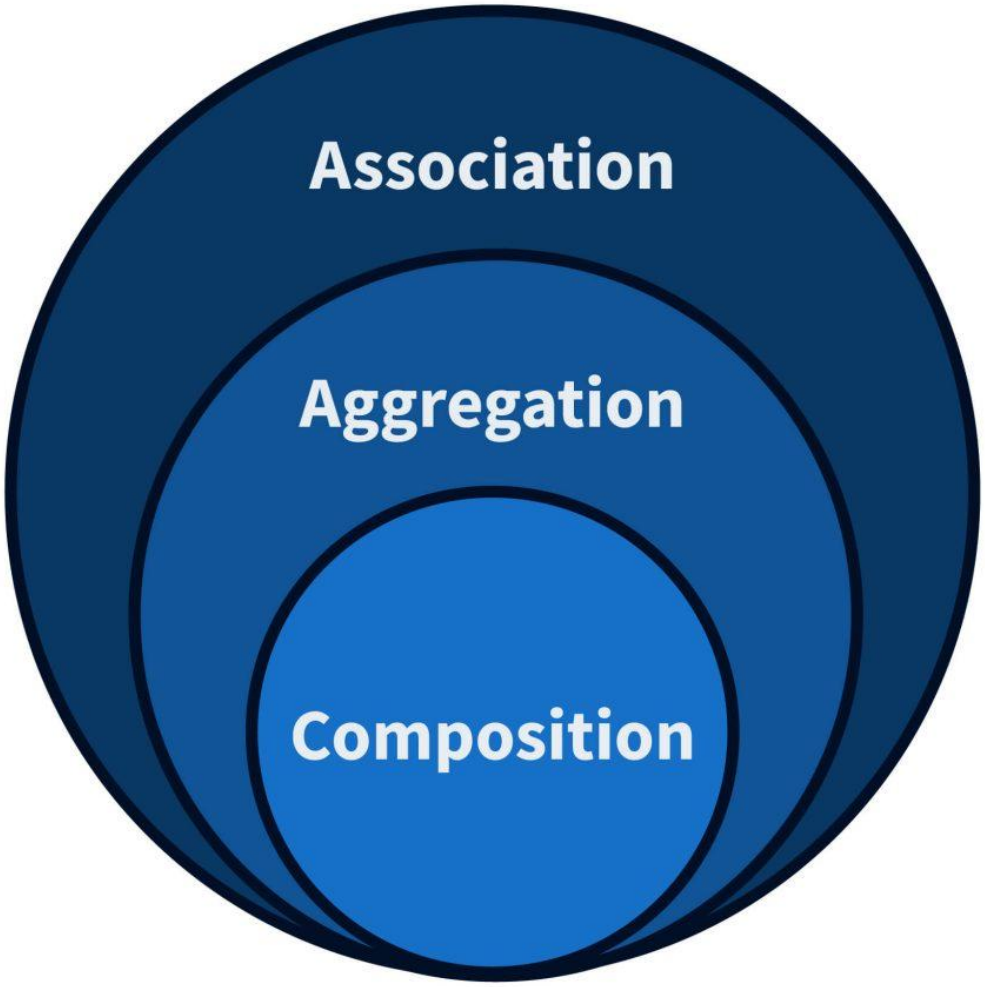
Association

Association shows the relationship between two separate classes which establishes through the objects of classes. As discussed above in message passing, that in OOPS an object can communicate to the other object to use their functionality or methods. Association shows “HAS-A” relationship between classes. So if we take the example of “Hotel Management System”, if Hotel is class to represent hotel and Room class represents room of hotel then we can show the association between them like this:



Now, Association can be of following types:

– one-to-one: In one-to-one association, only a single object is associated with another object at a time.



– one-to-many: In one-to-many association, one object is associated with one or more than one object at a time.

– many-to-one: In many to one association many objects are associated with a single object at a time.

– many-to-many: In many to many associations, many objects are associated with one or more than one object at a time.

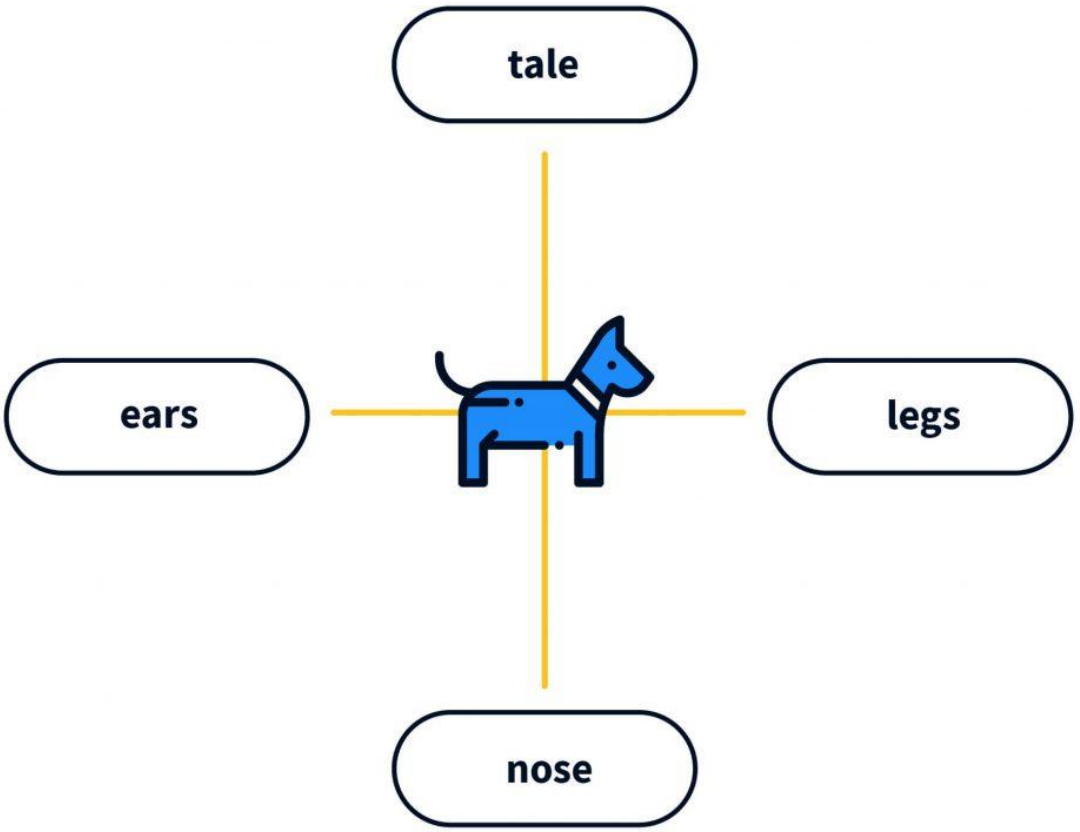
Aggregation and Composition are two types of Association. These two have been defined below in detail.

Composition

The “Composition” is a special form of “Association”. As shown in the above figure, the Composition circle is inside the Association circle so it must be having some restrictions over Association. So in Composition child class object is strongly dependent on the parent class object which means if the parent object is deleted then all child objects will also be deleted. In other words, child objects don’t have their own lifecycle. Composition is a stronger relationship. You can take a real-life example of a dog:

A dog has two legs, one tail, two ears, and one nose. So you can think of a dog as a parent class and legs, tail, ears, and nose as child classes of “Dog” class. So if you delete ears or nose class objects then the dog can survive, but if you delete the parent class object that is the dog then there is no point for legs, tail, ears, and nose class objects. So these child class objects will be deleted if the parent class object is deleted.

Note: Child class can exist independently but that child class object which is dependent on parent class object will be deleted if parent class object is deleted.



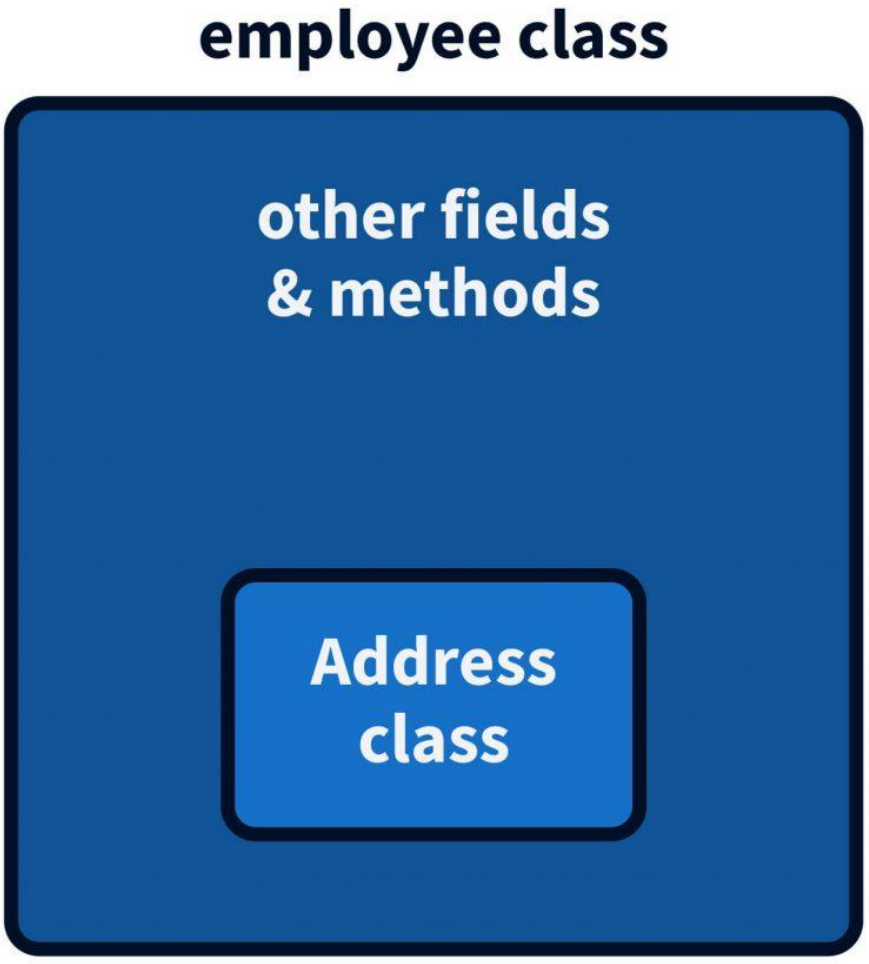
Aggregation

Aggregation is another form of Association. In Aggregation, child objects have their own lifecycle which means if the parent object is deleted then all child objects can exist independently. But there is child ownership in Aggregation which means if the parent object is deleted then the child object can exist independently but it can't belong to another parent class object. Aggregation is a weaker relationship. You can take a real-life example:

Suppose a person has 3 things: a computer, a gaming laptop, and a mouse. So, if that person gives his laptop to another person then also that laptop will remain of the first person or person who is the owner of that laptop which means ownership is there. Even if the parent class object is deleted, the child class object that is the laptop here can exist independently but it can't belong to another class or to the person to whom the owner of the laptop gave the laptop.

So you can say that aggregation is a process in which one class defines another class as an entity and it promotes reusability and represents the “HAS-A” relationship between classes.

Suppose there is a class of “Scaler employee” and which in turn have another class “Address” object which represents the address of Scaler employee so in this way you are reusing the member of address class in scaler employee class and it shows “has-a” relationship between the two classes.



Describe the Spring framework

The Spring framework is used to make enterprise applications using Java. Its central features can work with essentially any Java application.

Spring is an application framework and container that facilitates the inversion of control (IoC). With inversion of control, a generic network sends the flow of control to customized portions of a program.

Describe what makes a queue and a stack different

A stack is based on the principle of Last In, First Out (LIFO). A queue is different in that it's based on the First In, First Out (FIFO) principle.

How does garbage collection work in Java?

Garbage collection is used when an object is either not being referenced or isn't being used. At that point, garbage collection can automatically destroy the object.

What is the base class of all exception classes in Java?

The base class of all exception classes is Java.lang.Throwable. This being the base class means that all exception classes are derived from it.

What are the differences between the static and non-static methods?

With the static method, the static keyword has to precede the method name, and you call it using the class, as in "className.methodName." Also, you can't access any non-static methods or instance variables with the static method.

The non-static method doesn't require you to use a static keyword before the method name, and you can call it as you would any general method. And, the non-static method can access any other static method and variable without the need to rate an instance of the class.

Describe constructor chaining

Constructor chaining in Java involves calling a constructor from another in connection with the current object. To make this happen, a subclass constructor has to be able to invoke its constructor first.

The constructor chaining process can be accomplished either within the same class using "this()" or from the base class by using "super()."

How to Structure OOP Programs:

Let's take a real-life example of the Hotel Management system which we took in starting by extending it further and design an OOP program. Suppose you have to create software for a hotel that can handle all the rooms, their status, bookings, services, payments, then how will you create simple, reusable software to manage the hotel.

As there will be a lot of categories of rooms in the hotel and you have to maintain the status of each room and its booking and payments, so you can use classes and objects for creating the software.

You have to identify real-life objects in the given scenario and club all the object-related information in class and you can use that class by creating objects of that class.

In the given example here is how a programmer can think of creating classes:

– Room class: To manage rooms of the hotel
– Booking class: To manage all booking of the hotel
– Payment class: To manage all operations of payments of bookings
– Customer class: To manage all operations related to the customer who will come to the hotel to book a room.

Now classes are defined programmer can think of defining attributes of each class:

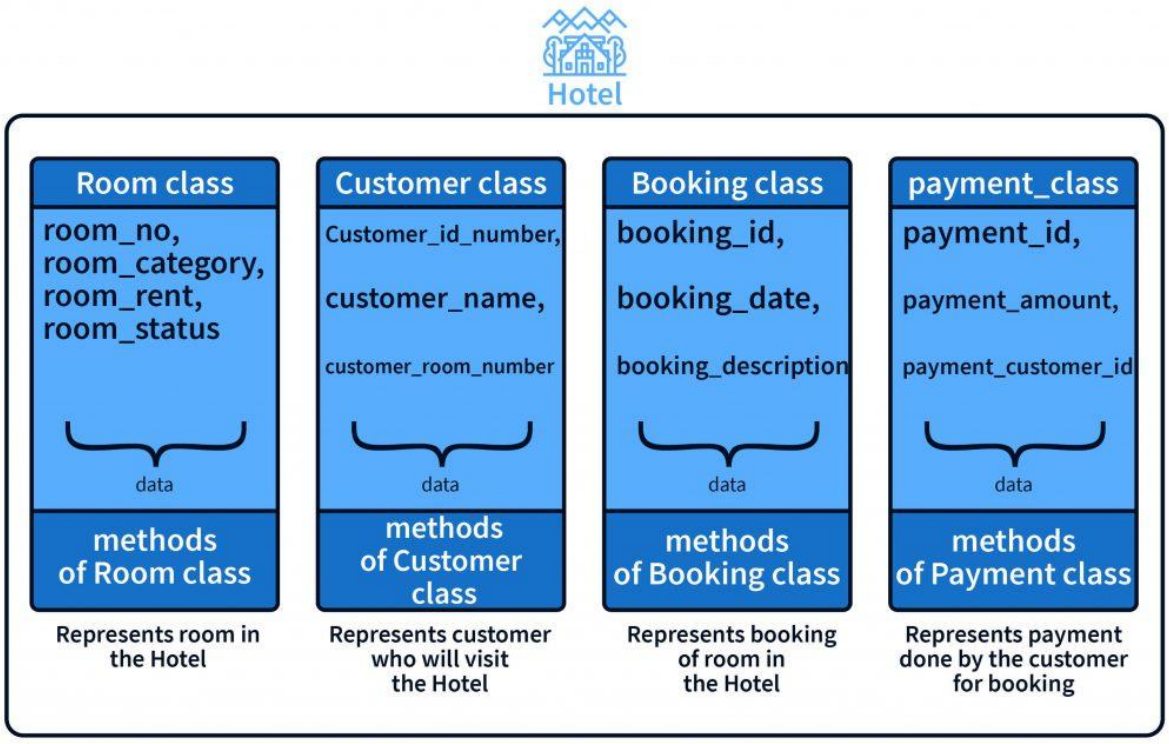
– Room class : room_no, room_category, room_status, room_rent

– Booking class: booking_id, booking_date, booking description
– Payment class : payment_id, payment_amount, payment_customer_id
– Customer class: customer_id_number, customer_name, customer_room_number

These all are the possible attributes which you can take in each class.

Now a programmer can think of adding methods or logic in each class according to his or her use cases or requirements. Suppose in Room class the programmer can add methods like this: checkIn(), checkOut(), getRoomStatus() and so on…

So here is a diagram representing the design of the OOP program in which you can group together related data(fields or variables) and behaviours(methods) to form a simple template or class:



Generics

Java generics are mainly used to impose type safety in programs. Type safety is when the compiler validates the datatype of constants, variables, and methods whether it is rightly assigned or not.

For example, we cannot initialize a variable as an integer and assign a string value to it. To avoid such mismatch between them, we can restrict the usage of a specific type of object only, like Integer, Float, etc.

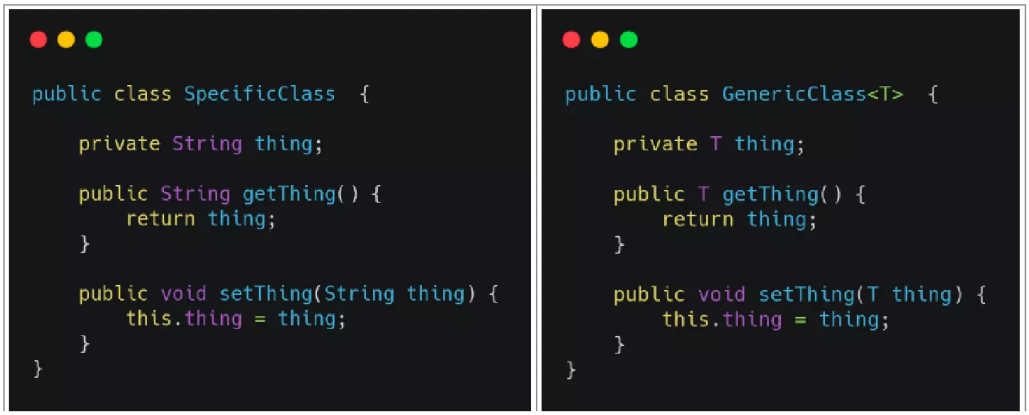
The following are some of the extensively used naming conventions for generic parameters:

E – Element (used extensively by the Java Collections Framework, for example, ArrayList, Set, etc.)
K – Key (Used in Map)
N – Number
T – Parameter Type
V – Value (Used in Map)

Types of Generics in Java

Java Generic Type Class

The real use of generics in its true sense can be observed in its application of creating classes. We can create a single class without specifying the data type of its object, in other words, using a generic data type we can create multiple classes with different data types. Consider the following two ways of setting and getting an item through a class:



```
public class SpecificClass  {

    private String thing;

    public String getThing() {
        return thing;
    }

    public void setThing(String thing) {
        this.thing = thing;
    }
}
```

```
public class GenericClass<T>  {

    private T thing;

    public T getThing() {
        return thing;
    }

    public void setThing(T thing) {
        this.thing = thing;
    }
}
```

On the left-hand side, we can see a class that is used to get and set a string object. This cannot be used to set any other type to the "thing". It makes the code quite rigid and not reusable.

Imagine if you had to create a new class for each kind of data type, it would be tedious. You would have to declare a new class for each kind of data type. On the other hand, the class on the right side is a generic class with a non-specific data type as an argument.

You can reuse it to pass any kind of object. All you have to do is specify the data type in the class object and just pass the right data:

```
public static void main(String args[]){
    //Creating Generic class object with String as parameter
    GenericClass<String> book= new GenericClass<>();
    book.set("Clean Code");

    //Creating Generic class object with Integer as parameter
    GenricClass<Integer> pricing = new GenericClass<>();
    pricing.set(150);
}
```

Java Generics Interface

Generics can be used in interfaces as well. The class implementing the interface can specify the type of parameter for the method. The following code snippet shows an interface where <T1, T2> are the generic parameters to the interface.

```
//Generic interface definition
   interface DemoInterface<T1,T2>{
   T1 doSomeOperation(T1 t);
   T2 doReverseOperation(T2 t);
   }
//Class implementing generic interface
class DemoClass implements DemoInterface<String,Integer>{
   public Integer doSomeOperation(String t){
      //logic
   }
   public Integer doReverseOperation(String t){
      //logic
   }
}
```

The class DemoClass implements the interface DemoInterface by passing String and Integer corresponding to T1's and T2's datatype respectively. Similarly, any number of classes implementing the interface can use different data types like Integer, Float, Objects, etc.

Generic Method

As the name suggests, the type can be passed through a method or constructor. We can declare a generic method that can be called with arguments of different types. Based on the type of parameter passed to the method, the compiler will handle the method call appropriately. If we don't want the whole class to be parameterized, we can make only generic methods.

```
 class GenericMethodDemo
{
 // A Generic method
```

```
static <T> void printGeneric (T element)
    {
      System.out.println(element);
    }
   public static void main(String[] args)
      {
      // Calling generic method with Integer as argument
      printGeneric(42);

      // Calling generic method with String as argument
      printGeneric("The answer is 42");
    }
}
```

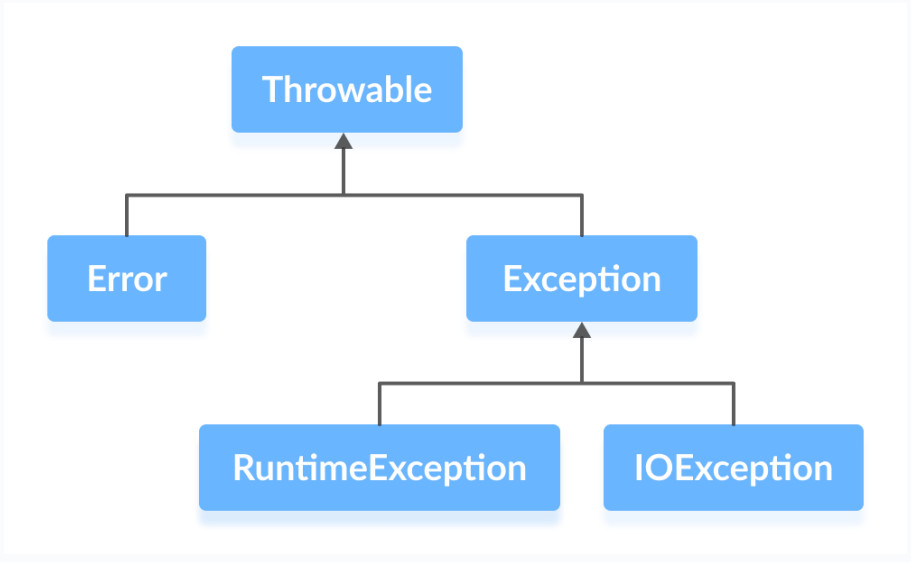
Exceptions

An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file
- Java Exception hierarchy

Here is a simplified diagram of the exception hierarchy in Java.



As you can see from the image above, the Throwable class is the root class in the hierarchy.

Note that the hierarchy splits into two branches: Error and Exception.

Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer and we should not try to handle errors.

Exceptions

Exceptions can be caught and handled by the program.

When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

Java Exception Types

The exception hierarchy also has two branches: RuntimeException and IOException.

1. RuntimeException

A runtime exception happens due to a programming error. They are also known as unchecked exceptions.

These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API - IllegalArgumentException
- Null pointer access (missing the initialization of a variable) - NullPointerException
- Out-of-bounds array access - ArrayIndexOutOfBoundsException
- Dividing a number by 0 - ArithmeticException

You can think about it in this way. “If it is a runtime exception, it is your fault”.

The NullPointerException would not have occurred if you had checked whether the variable was initialized or not before using it.

An ArrayIndexOutOfBoundsException would not have occurred if you tested the array index against the array bounds.

2. IOException

An IOException is also known as a checked exception. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

Trying to open a file that doesn’t exist results in FileNotFoundException

Trying to read past the end of a file

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

try...catch block

finally block

throw and throws keyword

1. Java try...catch block

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

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

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

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

Example: Exception handling using try...catch

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

    try {

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

```
    }

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

Output

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

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

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

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

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

2. Java finally block

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

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

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

Example: Java Exception Handling using finally block

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

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

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

Output

```
ArithmeticException => / by zero

This is the finally block
```

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

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

3. Java throw and throws keyword

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

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

Example: Exception handling using Java throw

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

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

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

Output

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

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

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

Example: Java throws keyword

```
import java.io.*;

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

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

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

Output

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

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

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

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

Reflection

In Java, reflection allows us to inspect and manipulate classes, interfaces, constructors, methods, and fields at run time.

There is a class in Java named Class that keeps all the information about objects and classes at runtime. The object of Class can be used to perform reflection.

Reflection of Java Classes

In order to reflect a Java class, we first need to create an object of Class.

And, using the object we can call various methods to get information about methods, fields, and constructors present in a class.

There exists three ways to create objects of Class:

1. Using forName() method

Here, the forName() method takes the name of the class to be reflected as its argument.

2. Using getClass() method

Here, we are using the object of the Dog class to create an object of Class.

3. Using .class extension

Now that we know how we can create objects of the Class. We can use this object to get information about the corresponding class at runtime.

Example:

```
import java.lang.Class;
import java.lang.reflect.*;

class Animal {
}

// put this class in different Dog.java file
public class Dog extends Animal {
  public void display() {
    System.out.println("I am a dog.");
  }
}

// put this in Main.java file
```

```
class Main {
  public static void main(String[] args) {
    try {
      // create an object of Dog
      Dog d1 = new Dog();

      // create an object of Class
      // using getClass()
      Class obj = d1.getClass();

      // get name of the class
      String name = obj.getName();
      System.out.println("Name: " + name);

      // get the access modifier of the class
      int modifier = obj.getModifiers();

      // convert the access modifier to string
      String mod = Modifier.toString(modifier);
      System.out.println("Modifier: " + mod);

      // get the superclass of Dog
      Class superClass = obj.getSuperclass();
      System.out.println("Superclass: " + superClass.getName());
    }

    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Output

```
Name: Dog
Modifier: public
Superclass: Animal
```

- obj.getName() - returns the name of the class
- obj.getModifiers() - returns the access modifier of the class
- obj.getSuperclass() - returns the super class of the class

Enums

In Java, an enum (short for enumeration) is a type that has a fixed set of constant values. We use the enum keyword to declare enums.

Example

```
enum Size {
 SMALL, MEDIUM, LARGE, EXTRALARGE
}

class Test {
 Size pizzaSize;
 public Test(Size pizzaSize) {
   this.pizzaSize = pizzaSize;
 }
 public void orderPizza() {
   switch(pizzaSize) {
     case SMALL:
       System.out.println("I ordered a small size pizza.");
       break;
     case MEDIUM:
       System.out.println("I ordered a medium size pizza.");
       break;
     default:
       System.out.println("I don't know which one to order.");
       break;
   }
```

```
 }
}

class Main {
 public static void main(String[] args) {
   Test t1 = new Test(Size.MEDIUM);
   t1.orderPizza();
 }
}
```

Output

```
I ordered a medium size pizza.
```

In the above program, we have created an enum type Size. We then declared a variable pizzaSize of the Size type.

Here, the variable pizzaSize can only be assigned with 4 values (SMALL, MEDIUM, LARGE, EXTRALARGE).

Notice the statement,

```
Test t1 = new Test(Size.MEDIUM);
```

It will call the Test() constructor inside the Test class. Now, the variable pizzaSize is assigned with the MEDIUM constant.

Based on the value, one of the cases of the switch case statement is executed.

Singleton Class

In Java, Singleton is a design pattern that ensures that a class can only have one object.

- To create a singleton class, a class must implement the following properties:
- Create a private constructor of the class to restrict object creation outside of the class.

- Create a private attribute of the class type that refers to the single object.

Create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

Singletons can be used while working with databases. They can be used to create a connection pool to access the database while reusing the same connection for all the clients. For example,

```
class Database {
  private static Database dbObject;

  private Database() {
  }

  public static Database getInstance() {

    // create object if it's not already created
    if(dbObject == null) {
      dbObject = new Database();
    }

     // returns the singleton object
    return dbObject;
  }

  public void getConnection() {
    System.out.println("You are now connected to the database.");
  }
}

class Main {
  public static void main(String[] args) {
    Database db1;
```

```
    // refers to the only object of Database
    db1= Database.getInstance();

    db1.getConnection();
  }
}
```

Output

```
You are now connected to the database.
```

In our above example,

- We have created a singleton class Database.
- The dbObject is a class type field. This will refer to the object of the class Database.
- The private constructor Database() prevents object creation outside of the class.
- The static class type method getInstance() returns the instance of the class to the outside world.
- In the Main class, we have class type variable db1. We are calling getInstance()using db1 to get the only object of the Database.
- The method getConnection() can only be accessed using the object of the Database.
- Since the Database can have only one object, all the clients can access the database through a single connection.

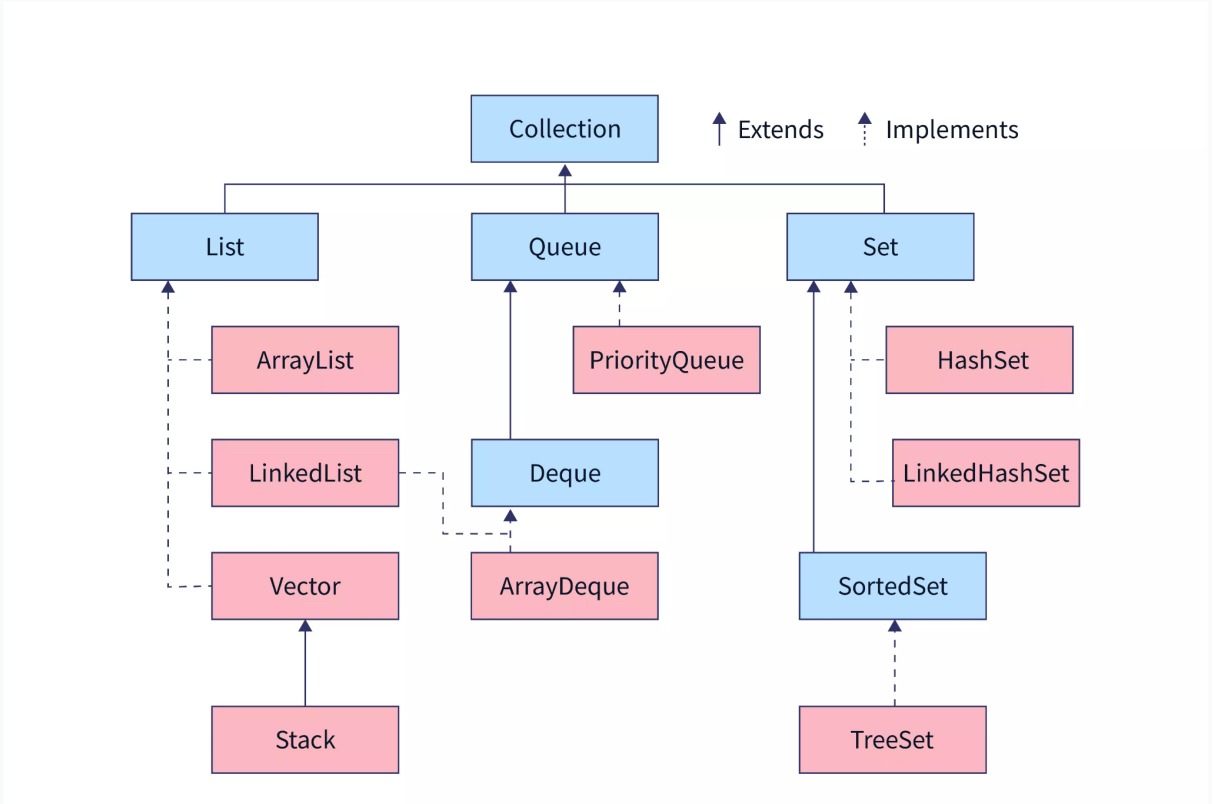
Collections

Consider the example of a piggy bank. We all had it during our childhood where we used to store our coins. This piggy bank is called as Collection and the coinsare nothing but objects. Technically, a collection is an object or a container that stores a group of other objects.

A Collection in Java is an object which represents a group of objects, known as its elements. It is a single unit. They are used to standardize the way in which objects are handled in the class.

Now that we know what Collection in Java is, let us try to understand the real-life use cases-

- Linked list emulates your browsing history, train coaches which are connected to each other etc.
- Stacks are like a stack of plates or trays in which the top-most one gets picked first.
- Queue is same as the real-life queues, the one who enters the queue first, it leaves it first too.



Autoboxing and unboxing

In autoboxing, the Java compiler automatically converts primitive types into their corresponding wrapper class objects.
In unboxing, the Java compiler automatically converts wrapper class objects into their corresponding primitive types.

```
import java.util.ArrayList;

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

    ArrayList<Integer> list = new ArrayList<>();

    //autoboxing
    list.add(5);
    list.add(6);

    System.out.println("ArrayList: " + list);

    // unboxing
    int a = list.get(0);
    System.out.println("Value at index 0: " + a);

  }
}
```

Output

```
ArrayList: [5, 6]
Value at index 0:
```

Enums

In Java, an enum (short for enumeration) is a type that has a fixed set of constant values. We use the `enum` keyword to declare enums.

```
enum Size {
  SMALL, MEDIUM, LARGE, EXTRALARGE
}
```

```
class Main {
   public static void main(String[] args) {
      System.out.println(Size.SMALL);
      System.out.println(Size.MEDIUM);
   }
}
```

Output

```
SMALL
MEDIUM
```