Java Notes Part 5

Inheritance In Java:

Real-World Meaning of Inheritance

- In real life, inheritance means **passing traits, properties, or wealth from parents to children**.

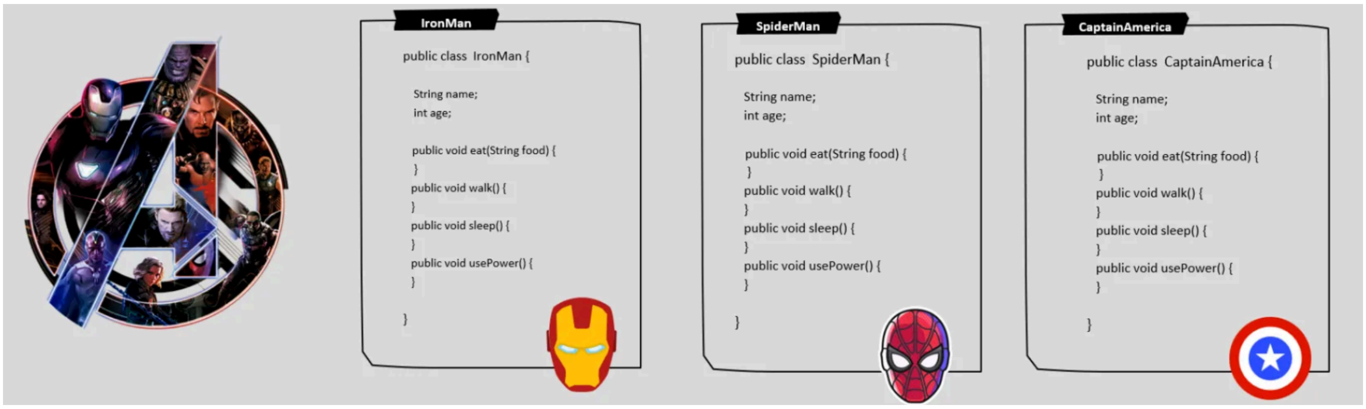
Example:

- Children inherit physical features from their parents.
- Family property is passed to the next generation.

Similarly, in programming, **one class can inherit properties and behaviors from another class**.

- In Java, inheritance is a mechanism that allows one class to inherit the properties and behavior of another class. Just like how a child inherits certain physical traits and characteristics from its parents, a child class in Java inherits certain properties and behavior from its parent class. This allows you to reuse code and create more efficient and organized class hierarchies.
- In object-oriented programming, **Inheritance** stands as a fundamental principle. It enables the formation of a new class by incorporating code from an already existing class. The newly formed class takes on the title of a subclass, while the original class is referred to as the superclass.
- The superclass holds the code that the subclass reuses and modifies as needed. This relationship is often described as the subclass inheriting from the superclass. The superclass is alternatively called a base class or parent class, while the subclass may be referred to as a derived class or child class.
- To understand inheritance, let's assume that you are trying to build Java classes representing various superheroes from marvel universe like shown below.
- If you see, there is a lot of duplicate code representing their name, age, how they eat, walk, sleep, and use power using variables & methods. The only method that may have a different implementation for each hero is how they use their power.
- To avoid duplicating the same code across multiple classes, you can create a parent class called Person that contains the shared properties and methods, and then inherit

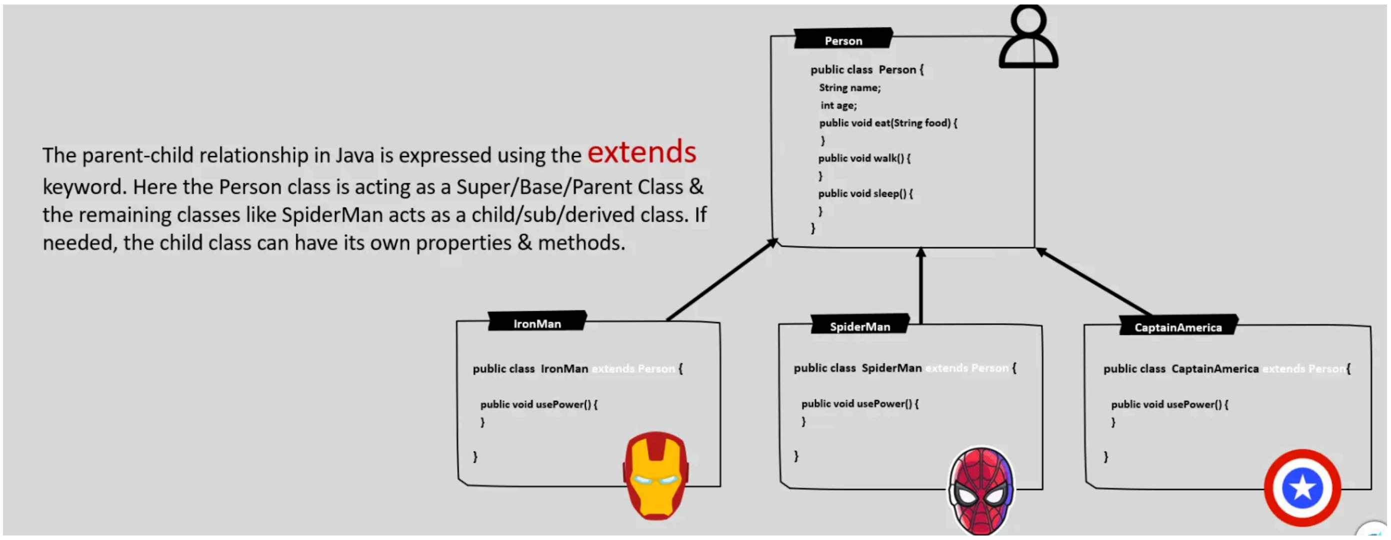
from that class to create subclasses for each superhero with their unique power implementation.



```
public class IronMan {

  String name;
  int age;

  public void eat(String food) {
  }
  public void walk() {
  }
  public void sleep() {
  }
  public void usePower() {
  }

}
```

```
public class SpiderMan {

  String name;
  int age;

  public void eat(String food) {
  }
  public void walk() {
  }
  public void sleep() {
  }
  public void usePower() {
  }

}
```

```
public class CaptainAmerica {

  String name;
  int age;

  public void eat(String food) {
  }
  public void walk() {
  }
  public void sleep() {
  }
  public void usePower() {
  }

}
```



The parent-child relationship in Java is expressed using the extends keyword. Here the Person class is acting as a Super/Base/Parent Class & the remaining classes like SpiderMan acts as a child/sub/derived class. If needed, the child class can have its own properties & methods.

```
public class Person {
  String name;
  int age;
  public void eat(String food) {
  }
  public void walk() {
  }
  public void sleep() {
  }
}
```

```
public class IronMan extends Person {

  public void usePower() {
  }

}
```

```
public class SpiderMan extends Person {

  public void usePower() {
  }

}
```

```
public class CaptainAmerica extends Person {

  public void usePower() {
  }

}
```

- **Super class/Base class/Parent class:** A class from which another class is derived.
- **Subclass/Derived class/Child class:** A class that is derived from a superclass.

Note: Using Inheritance we achieve the IS-A relationship in Java

Example**:**

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Syntax of Inheritance:

```
class SubClass extends SuperClass {
  // class body
}
```

Example1:

```
//parent class

class Animal {
        // methods and fields
}

//child class
class Dog extends Animal {
        // methods and fields of Animal are inherited
}
```

Example2:

Animal.java:

```
//parent class Animal.java
class Animal {
        // field and method of the parent class
        String name;

         public void eat() {
          System.out.println("Animal can eat");
        }

}
```

Dog.java:

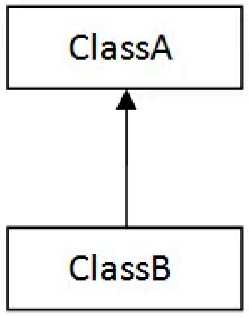
```
// child class inherits from parent// Dog.java
//Dog is an Animal
class Dog extends Animal {

        // new method in subclass
        public void bark() {
         System.out.println(name+ " Is barking..");
        }
}
```

<u>Demo.java:</u>

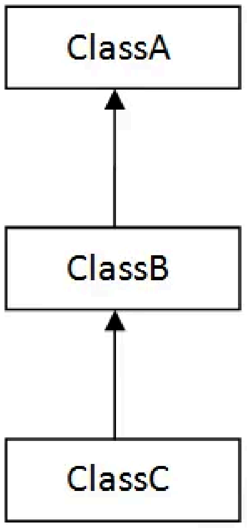
```
class Demo {

        public static void main(String[] args) {

                // create an object of the subclass
                Dog d1 = new Dog();

                // access field of superclass
                d1.name = "Tommy";
                d1.bark();

                // call method of superclass (inherited method)
                // using object of subclass
                d1.eat();
        }
}
```

Types of Inheritance:

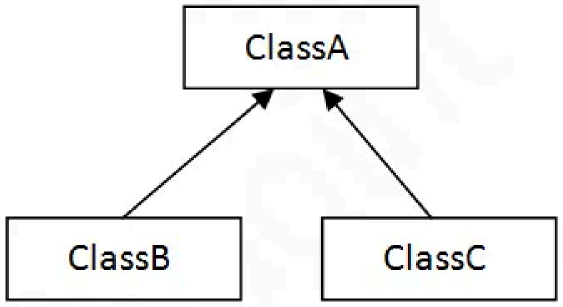
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance



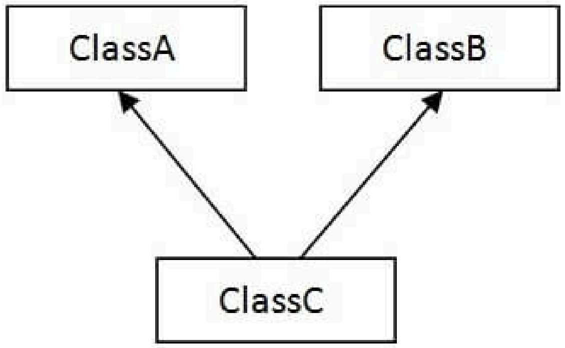
1) Single



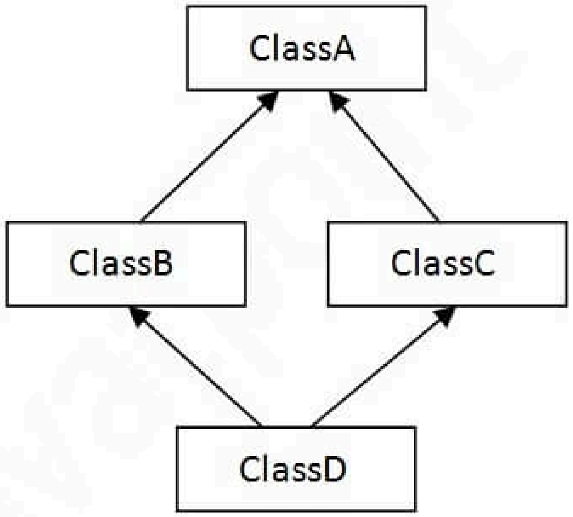
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

- Multiple inheritance and hybrid inheritance are not allowed in Java with classes because they can lead to several potential problems and complexities, such as the "**diamond**

problem," where there are conflicting implementations of a method from multiple parent classes.

- Based on class, there can be three types of inheritance in Java:
    1. Single Inheritance
    2. Multi-level Inheritance
    3. Hierarchical Inheritance.
- In Java, multiple and hybrid inheritance can be achieved using the **Interface** concept. We will learn about an interface later.

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

1. Single Inheritance

- One child inherits one parent.

    Dog extends **Animal**

Example:

```
class Animal{

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


 class Dog extends Animal{

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

```
class Demo{

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

2. Multilevel Inheritance

- When there is a chain of inheritance, it is known as *multilevel inheritance.*

Example:

```
class Animal{

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

class Dog extends Animal{

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

class BabyDog extends Dog{

        void weep(){
                        System.out.println("weeping...");
                }
}

 class Demo{

        public static void main(String args[]){

                BabyDog d=new BabyDog();
                d.weep();
                d.bark();
                d.eat();
        }
}
```

3. Hierarchical Inheritance:

- Multiple child classes inherit one parent.

```
      Animal
     /      \
   Dog      Cat
```

Example:

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

                Cat c=new Cat();
                c.meow();
                c.eat();
                //c.bark();//C.T.Error
        }
}
```

Multiple & Hybrid Inheritance in Java

- Java **does NOT support multiple inheritance with classes**.

Example not allowed:

```
class C extends A, B  // ERROR
```

Why is multiple inheritance not supported at the class level in Java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in Java.
- Consider a scenario where **A**, **B**, and **C** are three classes. The **C** class inherits the **A** and **B** classes. If **A** and **B** classes have the same method and you call it from a child class object, there will be ambiguity in calling the method of the **A** or **B** class.
- It will cause a diamond problem.

```
   A
  / \
 B   C
  \ /
   D
```

Example:

```
class A{

        void msg(){
                System.out.println("Hello");
        }
}

class B{

        void msg(){
                System.out.println("Welcome");
        }
}

class C extends A,B{//suppose if it were, compilation error

        public static void main(String args[]){
                 C obj=new C();
                 obj.msg();//Now which msg() method would be invoked?
        }
}
```

How is Multiple Inheritance Achieved?

- Using **Interfaces**, not classes.

What Does a Subclass Inherit from Its Superclass?

- In Java, a subclass does not inherit all aspects of its superclass. Instead, it selectively inherits the following:

Inherited:

- **Public members**
- **Protected members**
- **Default members (same package)**
- **Static members**

Not Inherited:

- **Private members**
- **Constructors**
- **Static blocks**
- **Instance blocks**

How does inheritance work in Java?

- Superclass constructor runs first.
- Consider the following example:

X.java:

```
package com.chitkara;
public class X {

        int i = 10;

        void funX() {
                System.out.println("inside funX() of X");
        }
}
```

Y.java

```
package com.chitkara;
public class Y extends X {

        int j = 20;

        void funY() {
                System.out.println("inside funY() of y");
        }

        public static void main(String[] args) {

                Y y1=new Y();
                System.out.println(y1.j);//Access subclass member
                System.out.println(y1.i); //Access inherited member

                y1.funY();
                y1.funX();
        }
}
```

Key Points:

1. **Default Constructor and `super();`**
    - Every Java class has a **constructor**, at least the default constructor.
    - The `super();` statement is implicitly added to call the superclass's constructor.
2. **Implicit `Object` Class Inheritance:**
    - If a class does not explicitly extend another class, it implicitly extends the `Object` class.
    - The `Object` class is part of the `java.lang` package.
    - This makes `Object` the root of the class hierarchy in Java.
3. **Constructor Calls:**
    - The object of the superclass is created first, followed by the object of the subclass.
    - The superclass's constructor is called using the `super();` statement in the subclass constructor.

Example Code for Constructor Behavior:

X.java:

```
package com.chitkara;
public class X {

        public X() {
                System.out.println("Inside the constructor of X class");
            }

    }
```

Y.java:

```
    package com.chitkara;
    public class Y extends X {

            public Y() {
                System.out.println("Inside the constructor of Y class");
            }


            public static void main(String[] args) {

                Y y1 = new Y();
            }
    }
```

Output:

```
Inside the constructor of the X class
Inside the constructor of the Y class
```

Object Class: Root of Java

- Every class in Java implicitly extends the **Object** class.
- This **Object** class belongs to **java.lang** package.

Example:

These are the same:

```
class Person {
        // Code for the Person class
}
```

And

```
class Person extends Object {
    // Code for the Person class
}
```

Important Notes:

1. **Object Creation Order:**
    - The superclass's object is created first before the subclass's object.
2. **Association Between Objects:**
    - The superclass's object is created in association with the subclass's object.

Methods of the Object Class:

- The `Object` class provides several important methods that are inherited by all classes in Java:

1. **`protected Object clone() throws CloneNotSupportedException`**
    - Creates and returns a copy of this object.
2. **`public boolean equals(Object obj)`**
    - Indicates whether another object is "equal to" this one.
3. **`protected void finalize() throws Throwable`**
    - Called by the garbage collector when no more references to the object exist.
4. **`public final Class getClass()`**
    - Returns the runtime class of the object.

5. `public int hashCode()`
    - Returns a hash code value for the object.
6. `public String toString()`
    - Returns a string representation of the object.
7. `public void notify()`
    - Wakes up a single thread waiting on this object’s monitor.
8. `public void notifyAll()`
    - Wakes up all threads waiting on this object’s monitor.
9. `public void wait()`
    - Causes the current thread to wait until another thread invokes `notify()` or `notifyAll()`.
10. `public void wait(long timeout)`
    - Causes the current thread to wait for a specified amount of time or until `notify()`/`notifyAll()` is called.
11. `public void wait(long timeout, int nanos)`
    - Causes the current thread to wait for a specified amount of time (with nanosecond precision) or until `notify()`/`notifyAll()` is called.

Dynamic or runtime polymorphism:

- In Java, polymorphism refers to the ability to perform a single action in multiple ways. The term "polymorphism" is derived from the Greek words "poly" and "morphs", where **"poly" means many** and **"morphs" means forms**, hence it means "many forms".
- In real life, we also see many examples of polymorphism. For instance, a woman can play multiple roles in a day, like a mother, a wife, an employee, and a sister. So the same person possesses different behavior in different situations. Polymorphism is considered one of the important features of Object-Oriented Programming.

- Similarly, in programming:
    - A single method behaves differently depending on the object.
- Based on the type of binding, there are two types of polymorphism in Java:
    1. **Compile-Time Polymorphism**

       Achieved using:

       - **Method Overloading**

       The decision happens at compile time.

    2. **Runtime Polymorphism**

       Achieved using:

       - **Method Overriding**

       The decision happens at runtime by the JVM.

       This is also called:

       - Dynamic polymorphism
       - Dynamic method dispatch

Method Overriding

- As we know, an object of a child class can also access the method of its parent class also. But, if the child class object does not satisfy with the implementation of the inherited method, the child class can re-implement the inherited method with its own implementation; this concept is known as Method Overriding in Java.
- If an overridden method is called inside the subclass methods, then the version defined in the subclass will always be called, and to access the version of super class in the subclass methods, we have to use the **super** keyword.
- When overriding a method in the subclass, use the **@Override** annotation with the method signature so that the compiler will check if the method signature in super class and subclass is the same or not. If not the same, then the compiler will report an error.

Usage of Java Method Overriding:

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

Example:

<u>A.java:</u>

```
package com.chitkara;
class A{

        void show(){
                System.out.println("Inside show of class A");
        }

}
```

<u>B.java:</u>

```
package com.chitkara;
class B extends A{

        @Override
        void show(){  //class B has overridden the show method of class A
                System.out.println("Inside show of class B");
        }

        void fun(){
                  show();
                  super.show();
        }

        public static void main(String args){

                 A a = new A();

                a.show();
                System.out.println("-=-=-=-=-=-=-=-=--=-=-=-=-=-=-=-=-=");

                B b = new B();
                b.show();

                System.out.println("-=-=-=-=-=-=-=-=--=-=-=-=-=-=-=-=-=");
                b.fun();
        }
}
```

Output:

```
Inside show of class A
=-=-=-=-=-=-=-=--=-=-=-=-=-=-=-=-=
Inside show of class B
=-=-=-=-=-=-=-=--=-=-=-=-=-=-=-=-=
Inside show of class B
Inside show of class A
```

Rules for Method Overriding in Java

1. **Same Method Signature**:
   - The overriding method in the child class must have the same name as the method in the parent class.
   - The parameter list (number, type, and order of parameters) must match exactly with the method in the parent class.
2. **Inheritance**:
   - The child class must inherit from the parent class, establishing an **IS-A** relationship.
3. **Access Modifier**:
   - The access modifier of the overriding method cannot be more restrictive than the method in the parent class. For example:
     - If the parent method is `public`, the overriding method cannot be `private or default`.
4. **Return Type**:
   - The return type of the overriding method must be the same or a subtype (covariant return type) of the return type declared in the parent class.
   - The primitive types are not allowed as covariant return types.

5. **Method cannot be `final`, `static`, or `private`**:
    - A `final` method cannot be overridden because it is immutable.
    - A `static` method belongs to the class, not the instance, so it is not subject to overriding but can be **hidden**.
    - A `private` method is not inherited and thus cannot be overridden.
6. **Exception Handling**:
    - The overriding method cannot throw checked exceptions that are broader than those declared in the parent method.
    - It can throw fewer or no exceptions, or unchecked exceptions.
7. **Annotation**:
    - It is a good practice to use the `@Override` annotation to ensure that the method is correctly overriding a superclass method.

Super Keyword in Java

- The **`super`** keyword in Java is used to refer to the immediate parent class's object.
- When an instance of a subclass is created, an instance of the parent class is implicitly created, which can be accessed using the `super` keyword.

Usage of the `super` Keyword in Java

1. **Access Parent Class Instance Variables**:
    - The `super` keyword can be used to refer to the instance variables of the parent class when they are hidden by subclass variables.
2. **Call Parent Class Methods**:
    - The `super` keyword can be used to invoke methods of the parent class if they are overridden in the subclass.
3. **Call Parent Class Constructor**:
    - The `super()` statement is used to invoke the constructor of the immediate parent class.

- It must be the first statement in the subclass constructor.

Example1: referring to the immediate parent class instance variable:

```
class Animal{
        String color="white";
}

class Dog extends Animal{

        String color="black";

        void printColor(){
                System.out.println(color);//prints color of Dog class
                System.out.println(super.color);//prints color of Animal class
        }
}


class Demo{

        public static void main(String args[]){
                Dog d=new Dog();
                d.printColor();
        }
}
```

Example2: referring to the immediate parent class instance method:

```
class Animal{

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

class Dog extends Animal{

        @Override
        void eat(){
                System.out.println("eating bread...");
        }

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

        void work(){

                eat();
                super.eat();
                bark();
        }
}


class Demo{

        public static void main(String args[]){
                Dog d=new Dog();
                d.work();
        }
}
```

Example3: invoking the parent class constructor.

```
class Animal {

        Animal(String name) {
                System.out.println("animal is created with name: " + name);
        }
}

class Dog extends Animal {

        Dog(String name) {
                super(name);
                System.out.println("dog is created with " + name);
        }
}


class Demo {
        public static void main(String args[]) {
            Dog d = new Dog("Tommy");
        }
    }
```

Note: The super keyword and the this keyword can not be used inside the static area.

Example of inheritance: Calling the constructor of the superclass

Vehicle.java: Base class

```
    public class Vehicle {

            private double basePrice;
            private double gstPercentage;

            public Vehicle(double basePrice, double gstPercentage) {
                this.basePrice = basePrice;
                this.gstPercentage = gstPercentage;
            }

            public double getBasePrice() {
                return basePrice;
            }

            public double getOnRoadPrice() {
                return basePrice + (basePrice * gstPercentage / 100.0);
            }
    }
```

LuxuryVehicle.java: Subclass

```
    public class LuxuryVehicle extends Vehicle {

        private double luxuryTaxPercentage;
      public LuxuryVehicle(double basePrice, double gstPercentage, double
      luxuryTaxPercentage) {
          super(basePrice, gstPercentage);
          this.luxuryTaxPercentage = luxuryTaxPercentage;
        }

        @Override
        public double getOnRoadPrice() {
          // Calculate the base on-road price and add the luxury tax
          double baseOnRoadPrice = super.getOnRoadPrice();
          return baseOnRoadPrice + (getBasePrice() * luxuryTaxPercentage / 100.0);
        }
    }
```

<u>Demo.java:</u>

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

        // Regular vehicle
        Vehicle car = new Vehicle(500000, 18.0); // Base price: 500,000, GST: 18%
        System.out.println("On-road price of the car: " + car.getOnRoadPrice());

        // Luxury vehicle
        LuxuryVehicle luxuryCar = new LuxuryVehicle(1000000, 18.0, 10.0); // Base price:
    1,000,000, GST: 18%, Luxury tax: 10%

      System.out.println("On-road price of the luxury car: " + luxuryCar.getOnRoadPrice());
      }
}
```

Dynamic method dispatch:

- It is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

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

Super class reference points to the subclass object:

- In Java, a parent class reference can point to the child class object.
- Generally, to any class reference variable, we can assign the following 3 things:
    1. Same class object
    2. It's child class object
    3. null (default value)

Example:

```
Parent p = new Parent();

Parent p = new Child();

Parent p = null;
```

Upcasting:

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

  Example:

```
class A{ //Parent class
        --
}
class B extends A{  //Child class
        --
}

A a=new B();//upcasting, this is only possible if the B class is a child class of A
```

Example: Dynamic Method dispatch

```
class Bike{

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

class Splendor extends Bike{

       @Override
      void run(){
            System.out.println("running safely for 60km");
      }
      public static void main(String args[]){

            Bike b = new Splendor();//upcasting
            b.run();
       }
}
```

Output:

```
running safely for 60km.
```

Explanation:

- In this example, we are creating two classes, Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of the Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime. Since method invocation is determined by the JVM, not the compiler, it is known as runtime polymorphism.

Another Example: Runtime polymorphism with multilevel inheritance

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


	class Dog extends Animal{

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


	class BabyDog extends Dog{

		void eat(){
			System.out.println("drinking milk");
		}

		public static void main(String args[]){

			Animal a1=new Animal();
			Animal a2=new Dog();
			Animal a3=new BabyDog();

				a1.eat();
				a2.eat();
				a3.eat();
			}
	}
```

Output:

```
eating
eating pedigree
drinking milk
```

Golden Rule:

- Reference type decides accessible methods.
- Object type decides method execution.

Object Down casting and the instanceof operator:

instanceof operator:

- The `instanceof` operator is used to check whether an object is an instance of a specific class or subclass. It returns `true` if the object is an instance of the specified class, and `false` otherwise.

Example:

```
class Animal {

        public static void main(String args[]) {

                Animal a = new Animal();
                System.out.println(a instanceof Animal);// true
                System.out.println(a instanceof Object);// true
        }
}
```

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

Example:

```
class Animal{
        --
}

class Dog extends Animal {// Dog inherits Animal

        public static void main(String args[]) {

                Dog d = new Dog();
                System.out.println(d instanceof Dog);// true
                System.out.println(d instanceof Animal);// true
        }
}
```

Object Down casting:

- As we know, to a parent class variable we can assign the child class object also, and from that parent class variable, if we try to call any overridden method, then due to Runtime polymorphism, the overridden method will be called. But if a parent class reference points to a child class object, with that parent class reference, we can not call the child class-specific methods, which are not available inside the parent class.
- To call the child class-specific method from the parent class reference variable, we need to downcast the parent class variable to the appropriate child class object.

Example:

```
class Animal {

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


class Dog extends Animal {

	@Override
	void eat() {
		System.out.println("eating bread...");
	}

	// specific method of child class
	void bark() {
		System.out.println("barking...");
	}
}

class Demo {

	public static void main(String args[]) {

		Animal parent = new Dog();
```

```
                parent.eat(); // eating bread...

                // calling child class specific method with parent class variable
                // parent.bark(); // C T Error

                // downcasting parent class variable to the child class object
                Dog d = (Dog) parent;
                d.bark();
        }
}
```

Note: We can downcast the parent class variable to the child class object only if the Parent class variable points to the Child class object; Otherwise, it will throw a runtime exception called *ClassCastException*.

Example:

```
class Animal {

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

class Dog extends Animal {

        @Override
        void eat() {
                System.out.println("eating bread...");
        }

        // specific method of child class
        void bark() {
                System.out.println("barking...");
        }
}

class Demo {

        void doSomething(Animal a) {
                a.eat();

                if (a instanceof Dog) {
                        Dog d = (Dog) a;
                        d.bark();
                }
        }

        public static void main(String args[]) {

                Demo d1 = new Demo();

                d1.doSomething(new Animal());
                d1.doSomething(new Dog());
        }
}
```

Variables Do Not Override in Java:

- In Java, **method overriding** is an important concept in inheritance and dynamic method dispatch. However, **variables (fields) do not support overriding**.
- Instead, variables follow a concept called **variable hiding**.
- Method calls are decided **at runtime** based on the object type (dynamic binding). But variable access is decided **at compile time** based on the reference type (static binding).

So:

- Methods: Runtime decision

- Variables: Compile-time decision

Example:

```
class A {
  int x = 10;
}
class B extends A {
  int x = 20;
}
public class Demo {
  public static void main(String[] args) {
        A obj = new B();
        System.out.println(obj.x);// 10
    }
}
```

The final keyword in Java:

- The **final keyword** in Java is used to ***restrict modification***. The Java final keyword can be used in many contexts. It can be applied to:

    1. Variable
    2. Method
    3. Class

- If you make any variable final, you cannot change the value of a final variable(It will become constant).
- In Java, the final variable must be initialized before we use it, either at the time of declaration or inside the constructor of the class.
- If you make any method final, you cannot override it inside the child class.
- If you make any class a final, you cannot extend it. The final class does not have the child class.

1. The final Variable:

- A `final` variable cannot be modified after it is initialized.

```
final int MAX = 100;

MAX = 200;   // // Compile-time error: cannot assign a value to a final variable
```

Initialization Rule

- A final variable must be initialized:

    1. At declaration, OR
    2. Inside constructor

Example:

```
class Test {
    final int x;
    Test() {
        x = 10; // initialized in constructor
    }
}
```

- After initialization, the value cannot change.

The final Reference Variable

- Important point:

```
final Student s = new Student();
```

- You cannot change the reference:

```
s = new Student(); // error
```

- But you **can modify object data**:

```
s.name = "Raj"; // allowed
```

- Final stops reference change, not object change.

2. The final Method

- A **final method cannot be overridden** in a child class.

Example:

```
class Parent {
    final void show() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {

    // Compilation error: cannot override the final method from Parent
    void show() {
        System.out.println("Child method");
    }
}
```

Reason:

- The parent wants the method behavior fixed.

3. The final Class:

- A `final` class cannot be subclassed.

Example:

```
final class FinalClass {
    // This class cannot be extended.
}

class Child extends FinalClass { // Compile-time error: cannot subclass final class

}
```

Many core Java classes are final for security.

Example:

String

Math

Wrapper classes

Overriding the toString() method of the Object class:

public String toString()

- The toString() method belongs to the **java.lang.Object** class. Provides a String representation of an object and is used to convert an object to a String. The default toString() method of the Object class returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@’, and the unsigned hexadecimal representation of the object's hash code. In other words, it is defined as:

```
// Default behavior of toString() is to print class name, then

// @, then the unsigned hexadecimal representation of the hash code of the
//object
public String toString() {

        return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Note: Whenever we try to print any Object reference, the toString() method is called internally.

- It is always recommended to override the toString() method to get our own String representation of an object to show the meaningful object data.

Example:

```
package com.chitkara;

class Student{

        private int rollno;
        private String name;
        private String city;

        Student(int rollno, String name, String city){
                this.rollno=rollno;
                this.name=name;
                this.city=city;
        }

         public static void main(String args[]){
                   Student s1=new Student(101,"Raj","lucknow");
                   Student s2=new Student(102,"Vijay","ghaziabad");

                   System.out.println(s1);//println method call s1.toString()
                   System.out.println(s2);//println method call s2.toString()
        }
}
```

Output:

```
        com.chitkara.Student@1fee6fc
        com.chitkara.Student@1eed786
```

- Let's override the toString() method from the Object class in our Student class.

Example:

```
package com.chitkara;
class Student {

        private int rollno;
        private String name;
        private String city;

        Student(int rollno, String name, String city) {
                this.rollno = rollno;
                this.name = name;
                this.city = city;
        }
        @Override
        public String toString() {// overriding the toString() method
                return rollno + " " + name + " " + city;
        }

        public static void main(String args[]) {

                Student s1 = new Student(101, "Raj", "lucknow");
                Student s2 = new Student(102, "Vijay", "ghaziabad");

                System.out.println(s1);
                System.out.println(s2);
        }
}
```

Output:

```
101 Raj lucknow

102 Vijay ghaziabad
```

Overriding the finalize() method of the Object class:

- `finalize()` is a method of the `Object` class called **before the object is garbage collected**.
- This method provides an opportunity for an object to release resources such as memory, file handles, or network connections before it is destroyed.

Purpose

- Used for cleanup tasks:
    - Closing files
    - Releasing resources
    - Network cleanup

Method Syntax:

```
protected void finalize() throws Throwable
```

Key Points:

1. **Definition**: The `finalize()` method is defined in the `Object` class.
   - It is automatically called by the garbage collector before an object is destroyed.
2. **Purpose**: It is mainly used for cleanup operations, such as releasing system resources or closing network connections.
3. **Override**: You can override the `finalize()` method in your own classes to define specific cleanup actions.
4. **Garbage Collection**: The `finalize()` method is not guaranteed to be called at any specific time, and there's no guarantee that it will be executed at all. It depends on the garbage collector.

Example:

```
package com.chitkara;
public class Demo {

        void fun1() {
                System.out.println("inside fun1 of Demo class");
        }

        @Override
        protected void finalize() throws Throwable {
                System.out.println("Cleanup operation is done");
                System.out.println("Object is destroyed by the GC");
        }

        public static void main(String[] args) {

                Demo d1 = new Demo();
                d1.fun1();
                d1 = null;
                System.gc(); // to invloke the GC manuall
        }
    }
```

- Here, we called the garbage collector explicitly by using System.gc(); initiated the memory cleanup process, otherwise the garbage collector is like a lazy person; if there is sufficient memory, it may not destroy the object immediately.

Modern Recommendation for the finalize() method:

- Not recommended in modern Java. Instead of finalize(), use:
- try-with-resources
- explicit cleanup methods

Method Hiding in Java:

- A class inherits all non-private static methods from its superclass. The act of **redefining** an inherited **static** method in a class is referred to as method hiding. In this context, the redefined static method in a subclass is said to hide the static method of its superclass. It is worth noting that when a non-static method is redefined in a class, this process is known as method overriding.

In Summary:

- Static methods belong to a class, not objects.
- If a subclass defines the same static method:
- It hides the parent method.

Example:

```
class Vehicle {

    static void start() {
        System.out.println("Vehicle starting");
    }
}

class Car extends Vehicle {

        static void start() {
                System.out.println("Car starting");
        }
}
```

Calling Methods:

```
        Vehicle.start(); // Vehicle method

        Car.start();     // Car method
```

Using Reference Variable:

```
        Vehicle v = new Car();

        v.start(); //Vehicle starting
```

- Because static methods are resolved at compile time, not at runtime.

Student Task:

Activity:

- Create two classes to represent an old TV and a smart TV.

Step 1: Create Parent Class

Create a class named `LgOldTV` with the following methods:

- `startTv()`: Starts the TV
- `stopTv()`: Stops the TV
- `increaseVolume()`: Increases the volume
- `changeChannel()`: Changes the channel in the old way

Step 2: Create Child Class

Create a child class named `LgSmartTV` that **extends** `LgOldTV`.

In this class:

1. **Override** the `changeChannel()` method so that the channel changes in a **smart way**.
2. Add a new method:
    - `playGame()`: Starts a game on the smart TV.

Step 3: Demonstrate Runtime Polymorphism

In the main method:

1. Create an object using a **parent reference and a child object**:

```
LgOldTV oldRemote = new LgSmartTV();
```

2. Call all applicable methods using this reference.

3. Also demonstrate how to call the smart TV-specific method.

Activity 2:

- Consider the following class:

Chef.java:

```
class Chef {

        String name;

        Chef(String name) {
                this.name = name;
                System.out.println("Chef " + name + " enters kitchen.");
        }

        void cookDish() {
                System.out.println(name + " cooks normal food.");
        }
}
```

- Create a child class of this Chef class as the **MasterChef** class
- Override the cookDish() method with
  - System.out.println(name + " cooks with special MasterChef style!");

- Define the following specific method inside the **MasterChef** class:

```
        void createSpecialDish() {
                System.out.println(name + " creates a signature dish!");
        }
```

- Inside the main method of the Demo class, create a MasterChef class object by supplying the name of the Chef, and store that object in the Chef class variable.

  Chef variable = MasterChef object.

- Call the methods:
  - **cookDish();**
  - **createSpecialDish();**

Packages in Java:

- A **package in Java** is a mechanism used to group **related classes, interfaces, enums, and annotations** into a single logical unit.

  Simply put:

  Package = Folder that organizes related classes

Why Do We Use Packages?

1. Code Organization

- Packages organize large projects into logical units, making maintenance easier.

2. Encapsulation

- Packages allow grouping related functionality together.

3. Access Control

- Access modifiers can restrict class and member visibility across packages.

4. Namespacing

- Packages prevent naming conflicts.

Example:

college.staff.cse.Employee

college.staff.ee.Employee

- Both classes are named Employee but belong to different packages.

Default Package

- If no package is specified:

```
class Demo {
}
```

- The above class Demo goes into the **default (unnamed) package**.
- Default packages are suitable only for:
    - Small programs
    - Testing
    - Temporary code
- Professional projects should always use named packages.

Common Java Packages

Package	Purpose
java.lang	Core classes (String, Object, System, Math)
java.util	Utilities (Scanner, List, Map, Collections)
java.io	Input/output operations
java.net	Networking
java.time	Date & time utilities

Package Structure

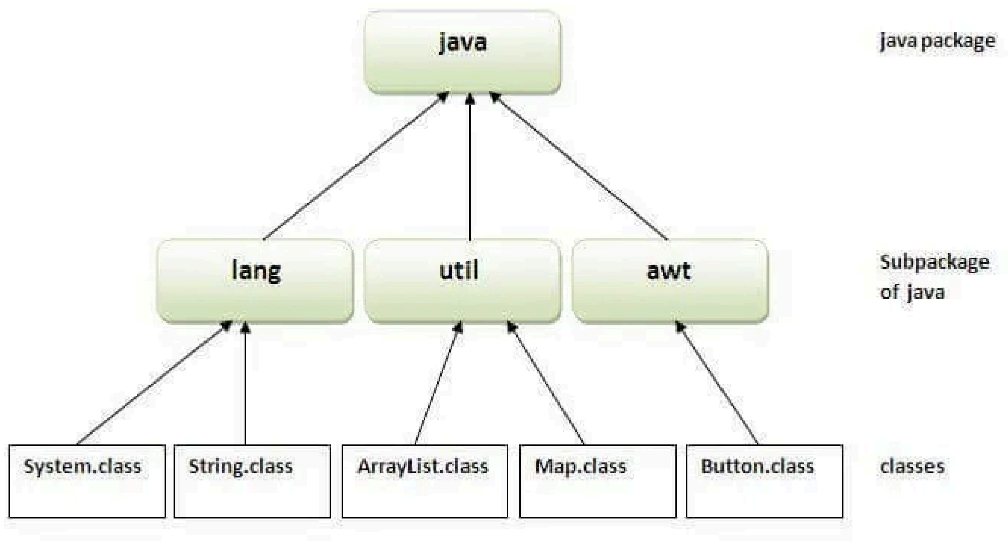
Example package:

```
college.staff.cse
```

Folder structure:

```
college/
    staff/
        cse/
```

Note: A Package is like a folder in a file directory. In Java, every package is a folder, but not every folder is a package



Package Naming Convention

- To avoid conflicts, developers follow naming standards:

Rules

1. Package names are lowercase.
2. Start with a reversed domain name.

Example:

com.oracle.project

com.google.auth

com.masai.studentapp

Company Structure Example:

com.company.department.project

Example:

com.oracle.sales.crm

Accessing Package Members

- To use classes from another package:

Method 1: Fully Qualified Name

```
java.util.Scanner sc = new java.util.Scanner(System.in);
```

Method 2: Import Specific Class

```
import java.util.Scanner;
```

Method 3: Import Entire Package

```
import java.util.*;
```

Note: The above import will just import classes only, NOT sub-packages.

The Java.lang Package

- This package is automatically imported.
- Classes available without import:

Example:

    - String
    - Object
    - System
    - Math
    - Wrapper classes (Integer, Double, etc.)

Static Import Statement:

- Static import allows direct use of static members without the class name.

Without Static Import:

```
double radius = 5.0;

double c = 2 * Math.PI * radius;
```

With Static Import:

```
import static java.lang.Math.PI;

double radius = 5.0;

double c = 2 * PI * radius;
```

Advantage

- Less typing for frequent static access.

Disadvantage

- Overuse reduces readability.

User-Defined Packages

- User-defined packages are those that are developed by users in order to group related classes, interfaces, and sub-packages.

- As a Java developer, we should keep our user-defined classes, interfaces, Enums, and annotations always inside a package.
- To create a package, use the `package` keyword: It should be the first statement of any Java application.

Example:

```
//Simple.java

package mypack;

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

Note: If a class is inside any package, in order to compile and run that class from the terminal (command prompt), we need to make use of the following command:

```
//to compile the above class
javac -d . Simple.java
//here after -d the .(dot) represents the current folder where we want to generate the byte code

//to run the above code
java mypack.Simple
//here we need to give the fully qualified class name
```

Sub-Packages

- Packages can be subdivided.

  Example:

        com.masai.model

        com.masai.service

        com.masai.controller

com.masai.utility

- Used in layered application architecture.

Example:

Simple.java

```
package com.masai.core;
public class Simple {

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

- To compile and run the above class using the command prompt:

```
//To compile the above class,

javac -d . Simple.java


//To run the above class

java com.masai.core.Simple
```

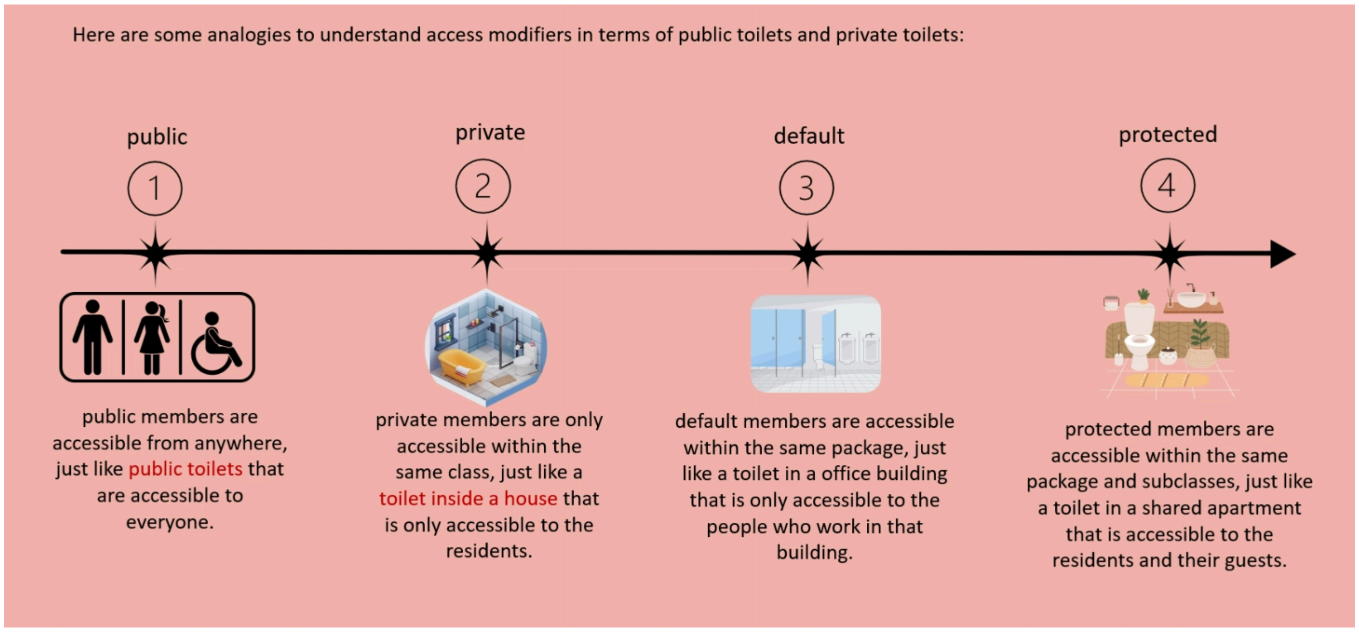
Access Modifiers in Java:

- The Access modifiers in Java specify the accessibility/visibility or scope of a variable, method, constructor, class, or interface. We can change the access level of variables, constructors, methods, and classes by applying the access modifier to them.

There are four types of Java access modifiers:

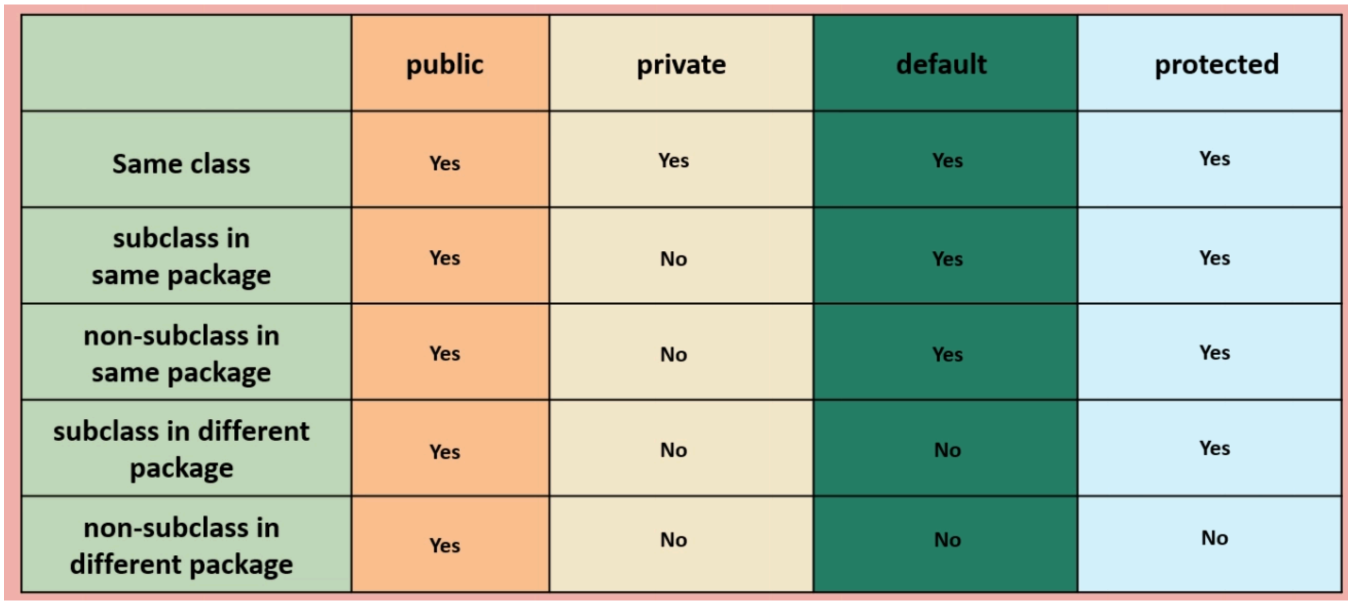
1. **private**: The access level of a private modifier is only **within** the class. It cannot be accessed from **outside** the class.
2. **default**: The access level of a default modifier is only within the **same package**. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **protected**: It is similar to default. The access level of a protected modifier is within the same package and outside the package through a child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package, or outside the package.

Note: An outer class can only be default or public, whereas class members can be public, private, protected, or default.


Here are some analogies to understand access modifiers in terms of public toilets and private toilets:
public
1
private
2
default
3
protected
4
public members are accessible from anywhere, just like public toilets that are accessible to everyone.
private members are only accessible within the same class, just like a toilet inside a house that is only accessible to the residents.
default members are accessible within the same package, just like a toilet in a office building that is only accessible to the people who work in that building.
protected members are accessible within the same package and subclasses, just like a toilet in a shared apartment that is accessible to the residents and their guests.

The following table displays the access levels for the different modifiers in Java:

```
}
```



	public	private	default	protected
Same class	Yes	Yes	Yes	Yes
subclass in same package	Yes	No	Yes	Yes
non-subclass in same package	Yes	No	Yes	Yes
subclass in different package	Yes	No	No	Yes
non-subclass in different package	Yes	No	No	No

Example: Role of Private Constructor:

- If you make any class constructor private, you cannot create an instance of that class from outside the class.
- Even you can not extend that class.

Example:

```
class A {
        private A() {// private constructor
        }

        void msg() {
                System.out.println("Hello java");
        }
}

public class Simple // extends A //ERROR  {
        public static void main(String args[]) {
                A obj = new A();// Compile Time Error
        }
}
```

Example of default access modifier:

- In this example, we have created two packages: **pack** and **mypack**. We are accessing the **A** class from outside its package, since the **A** class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{

 void msg(){
                    System.out.println("Hello");
          }
}


//save by B.java
package mypack;
import pack.*;
class B{

 public static void main(String args[]){

            A obj = new A();//Compile Time Error
            obj.msg();//Compile Time Error

          }
}
```

Example of protected access modifier:

- In this example, we have created the two packages **pack** and **mypack**. The **A** class of the **pack** package is public, so it can be accessed from outside the package. But the msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
```

```
public class A{

        protected void msg(){
                System.out.println("Hello");
        }
}

//save by B.java
package mypack;
import pack.*;
class B extends A{
         public static void main(String args[]){

          B obj = new B();
          obj.msg();

        }
}
```

Method overriding rule with access modifier:

- If you are overriding any method (declared in a subclass) must not be more restrictive.

Example:

```
class A{

        public void  msg(){
                System.out.println("Hello java");
        }
}

class Simple extends A{
        @Override
        void msg(){
                System.out.println("Hello java");
                }//C.T.Error

        public static void main(String args[]){
                Simple obj=new Simple();
                obj.msg();
        }
}
```

Abstraction in Java:

- **Abstraction** is one of the four main concepts of **Object-Oriented Programming (OOP)**:
    1. Encapsulation
    2. Inheritance
    3. Polymorphism
    4. Abstraction

Definition

- Abstraction means hiding internal implementation details and showing only the necessary functionality to the user.

In simple words:

Focus on **what an object does**, not **how it does it**.

The user uses features without knowing the internal logic.

Why Do We Need Abstraction?

- Abstraction helps in:
    1. **Hides Implementation Details**: Abstraction hides the internal mechanisms and only reveals the operations that are relevant to the user.
    2. **Simplifies Complexity**: By only providing the essential details, abstraction reduces complexity and simplifies coding.
    3. **Enhances Security**: By hiding data and restricting access to certain parts of code, abstraction helps secure the system.

4. **Improves Code Maintainability**: Changes to the internal implementation do not affect the user as long as the interface remains unchanged.

Real-Life Example: ATM Machine:

When using an ATM:

You can:

- Withdraw money
- Check balance
- Deposit money

But you **do not know**:

- How bank server communication happens
- How account validation occurs
- How transactions are processed

You only see options and results.

This is an abstraction.

How Abstraction is Achieved in Java?

- In Java, abstraction can be achieved in three ways:

1. Using a private access modifier

2. Using Abstract Class (Partial abstraction)

3. Using Interface (Full abstraction)

Abstraction using private methods

- Private methods hide implementation details inside the class.

Example

Account.java

```
public class Account {

	public void doOperation(int choice) {

		if (choice == 1) {
			withdrawAmount();
		} else if (choice == 2) {
			depositAmount();
		} else {
			System.out.println("Invalid choice");
		}
	}

	private void withdrawAmount() {
		System.out.println("Amount withdrawn successfully");
	}

	private void depositAmount() {
		System.out.println("Amount deposited successfully");
	}
}
```

Demo.java

```
import java.util.Scanner;
public class Demo {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);

		System.out.println("Enter choice:");
		int choice = sc.nextInt();
```

```
                Account account = new Account();
                account.doOperation(choice);
            }
        }
```

How does the above example show abstraction?

- Since Abstraction is about **hiding unnecessary details** and showing only what is necessary

1. **Hiding Implementation Details:**
    - The `withdrawAmount()` and `depositAmount()` methods are **private**.
    - The `doOperation` method acts as an interface to invoke these private methods based on the user's choice.
    - The user of the `Account` class does not know or need to know how these methods work internally.
2. **Providing Essential Features Only:**
    - The `doOperation` method provides a simple and clear way to perform operations without exposing the implementation logic.
    - This makes the `Account` class easy to use while keeping the internal workings secure and hidden.

- **Method-level abstraction** is achieved using **private methods**, which hide internal method implementation from users.
- **Class-level abstraction** is achieved using:
    - **Abstract classes** (partial abstraction)
    - **Interfaces** (conceptually full abstraction)

In short:

Private methods hide method implementation, while abstract classes and interfaces hide class-level implementation details.

Abstract Class in Java:

- Sometimes, you might design a Java class to represent a **concept** rather than representing tangible **objects**. Consider the scenario of developing classes for various educational subjects. A **subject** is an abstract concept; it doesn't have a physical existence. If someone asks you to provide details about a subject, your initial inquiry might be, “Which subject are you referring to?” It makes sense to discuss specific subjects like **mathematics** or **history**.
- In Java, you can create a class for which objects cannot be instantiated; its sole purpose is to represent an abstract idea shared among objects of other classes. Such a class is termed an abstract class. Conversely, a “concrete class" is one that is not abstract, and instances of it can be created. Up until now, all the classes you've created have been concrete classes.
- Abstract classes are designed to be extended by subclasses. They are particularly useful when you want to enforce a common structure or behavior across related classes while leaving some methods or properties to be defined in the subclasses.

Syntax:

```
public abstract class Subject {
        // other code
}
```

- As the **Subject** class is marked as abstract, creating an object of this class is not permitted, despite having a public constructor (that gets automatically added by the compiler).
- However, you can declare a variable of an abstract class, similar to how you declare variables for concrete classes.

Example:

```
Subject sub; // Compiles successfully
Subject sub = new Subject(); // Compilation fails
```

Note: To the variable of an abstract class, only 2 values can be assigned:

1. It’s child class object
2. null (default value)

Example:

```
Subject sub = new Mathematics();
```

Features of an Abstract Class:

1. **Can Have Concrete Methods:**
   - Abstract classes can include fully implemented (concrete) methods alongside abstract (unimplemented) methods.
2. **Can Have Variables and Constructors:**
   - Abstract classes can define variables and even have constructors. However, the constructor is invoked only when a subclass object is created.
3. **Cannot Be Final:**
   - Since an abstract class is meant to be extended, it cannot be declared as `final`. Declaring it as `final` would prevent subclassing, rendering the abstract class meaningless.
4. **Can Be Empty:**
   - An abstract class can be an empty class, defined only as a placeholder for subclasses.
5. **Requires Subclass Implementation:**
   - An abstract class must be extended by a child class to provide implementations for its abstract methods. **Without a subclass, an abstract class has no practical use.**

Key Differences: Abstract Class vs Concrete Class

Feature	Abstract Class	Concrete Class
Object creation	Can not be created directly using	Can be created using the new

Feature	Abstract Class	Concrete Class
	the new keyword	keyword.
Purpose	Represents an abstract idea or a concept.	Represents a complete, tangible object.
Methods	Can have both abstract and concrete methods.	Only concrete methods are allowed.
Usage	Used to define a base for subclasses.	Used to create objects directly.
Example	public abstract class Subject	public class Mathematics
final keyword	Abstract class can not be final	A concrete class can be final

Constructor Behavior in Abstract Class:

- Abstract class constructors are not used to create objects directly, but they run when a subclass object is created.

Rules of Abstract Class

- Cannot create object
- Can have a constructor
- Can have abstract & concrete methods
- Must be extended
- Cannot be final

Use Abstract Class When:

- Classes share behavior
- Need common fields
- Need partial implementation

Abstract Method:

- An **abstract method** is a method that is declared without implementation. Abstract methods are inherently incomplete and must be implemented by subclasses.
- **An abstract method:**
  - It is declared using the `abstract` keyword.
  - Does not have a method body (no `{}` block).
  - It can only exist inside an **abstract class** or inside an **interface**.

Example:

```
public abstract void pay(double amount);
```

Important Rule

Abstract methods:

- Cannot be private
- Cannot be final
- Cannot be static

Because subclasses must override them.

Note: inside a concrete class, we can not have an abstract method. Only an Abstract class or an Interface can have an abstract method.

Example: Payment System

- Payment is a common concept:
  - Credit Card

    - UPI
    - Net Banking
- Each payment type works differently.

<u>**Payment.java**</u> (Abstract Class)

```
public abstract class Payment {

	public abstract void pay(double amount);

	public void paymentStarted() {
		System.out.println("Payment process started...");
	}
}
```

<u>**UPIPayment.java**</u>

```
public class UPIPayment extends Payment {

	@Override
	public void pay(double amount) {
		System.out.println("Paid " + amount + " using UPI.");
	}
}
```

<u>**CreditCardPayment.java:**</u>

```
public class CreditCardPayment extends Payment {

	@Override
	public void pay(double amount) {
		System.out.println("Paid " + amount + " using Credit Card.");
	}
}
```

Demo.java:

```
public class Demo {

    public static void main(String[] args) {

        Payment payment = new UPIPayment();
        payment.paymentStarted();
        payment.pay(500);
    }
}
```

Output:

```
Payment process started...

Paid 500 using UPI.
```

Note: Subclasses that extend an abstract class must provide an implementation for all the abstract methods of the parent class, otherwise we need to mark the child class also as an abstract class.

Difference Between Abstract Methods and Concrete Methods

Feature	Abstract Method	Concrete Method
Defination	Declared but not implemented.	Declared and implemented.
Purpose	To enforce subclass implementation.	Provides functionality directly.
Body	No method body.	Must have a method body.
Usage	Used to define a base for subclasses.	Used to create objects directly.

Feature	**Abstract Method**	**Concrete Method**
Example	public abstract void displayDetails();	public void getName() { ... }
keywords	final, static, and private keywords are not allowed inside the abstract method.	Concrete methods can be final, static, or private.

Student Activity: Ride Booking System using Abstract Class

- Implement Ride Booking using an Abstract Class in Java

Real-World Scenario

- Ride booking applications like Uber or Ola provide multiple ride options:
    - Bike
    - Auto
    - Cab
- All rides share common operations:
    - Start ride
    - Calculate fare
    - End ride
- However, **fare calculation differs** for each ride type.
- To solve this, we create a common abstract class.

Problem Statement

Create a ride booking system where:

1. A base abstract class `Ride` defines common behavior.
    - A variable: double distance, which is initialized using the constructor inside the Ride class

- The following methods:
    - void startRide(): “Ride is Started.”
    - void endRide(): “Ride ended.”
    - abstract double calculateFare()

2. BikeRide, AutoRide, and CabRide classes implement their own fare calculation.
    - Bike: distance * 50;
    - Auto: distance * 80;
    - Cab: distance * 150;

3. A method `void bookRide(Ride ride)` of the Demo class performs booking using abstraction.

Solution:

Ride.java:

```
package com.chitkara;
public abstract class Ride {

	double distance;

	Ride(double distance) {
		this.distance = distance;
	}

	abstract double calculateFare();

	void startRide() {
		System.out.println("Ride started...");
	}

	void endRide() {
        System.out.println("Ride ended.");
    }
}
```

BikeRide.java

```
package com.chitkara;
public class BikeRide extends Ride {

    BikeRide(double distance) {
        super(distance);
    }

    double calculateFare() {
        return distance * 50;
    }
}
```

AutoRide.java

```
package com.chitkara;
public class AutoRide extends Ride {

    AutoRide(double distance) {
        super(distance);
    }

    double calculateFare() {
        return distance * 80;
    }
}
```

CabRide.java

```
package com.chitkara;
public class CabRide extends Ride {

    CabRide(double distance) {
        super(distance);
    }

    double calculateFare() {
        return distance * 150;
        }
    }
```

Demo.java:

```
package com.chitkara;
public class Demo {

    // Common booking method
    public void bookRide(Ride ride) {

        if (ride != null) {

            ride.startRide();
            double fare = ride.calculateFare();
            System.out.println("Total Fare: " + fare);
            ride.endRide();

        } else {
            System.out.println("Ride is null: please choose a proper ride");
        }
    }
    public static void main(String[] args) {

        Demo d1 = new Demo();

        Ride ride1 = new BikeRide(10);
        d1.bookRide(ride1);

        System.out.println();

        Ride ride2 = new CabRide(10);
        d1.bookRide(ride2);
    }
}
```

Student Task:

1. **Predict the output:**

```
class A {
        int x = 10;
}
class B extends A {
        int x = 20;
}
public class Test {
        public static void main(String[] args) {
                A obj = new B();
                System.out.println(obj.x);
        }
}
```

Output?

A) 10
B) 20
C) Compile error
D) Runtime error

2. **Predict the output**

```
abstract class Animal {
  public abstract void sound();
}
class Dog extends Animal {
  void sound() {
     System.out.println("Bark");
   }
}
public class Test {
   public static void main(String[] args) {
     Animal a = new Dog();
     a.sound();
   }
}
```

Output?

A) Bark
B) Animal
C) Compile error
D) Runtime error

Interface in Java:

What is an Interface in Java?

- An **interface** in Java is like a **contract** or a **blueprint** that defines **what a class must do**, but **not how it does it**.
- It contains **abstract method declarations** (without implementation) and **constants** that classes must implement.

Simple Definition

- An interface specifies **behavior** that a class promises to implement.

A Java Interface also represents the IS-A relationship.

Why Do We Need Interfaces?

- Interfaces are mainly used for:

  Achieving 100% Abstraction

  - Hide implementation details.

  Multiple Inheritance

  - A class can implement **multiple interfaces**.

  Loose Coupling

- Code depends on behavior, not implementation.

Standardization

- Many classes follow the same rules.

Syntax:

```
interface InterfaceName {
        // constants
        // abstract methods
}
```

Example:

<u>Printer.java</u>

```
interface Printer {
   void print();
}
```

- We also save an interface with the **.java** extension. And once we compile this .java file, a **.class** file is created by the **Java compiler** for the interface.

Note: The Java compiler adds **public** and **abstract** keywords before the methods defined inside an interface. Moreover, it adds **public**, **static,** and **final** keywords before variables inside an interface.

Example:

<u>Printer.java:</u>

```
public interface Printer{
        int number=10;
        void print();

    }
```

- Java compiler converts it as follows:

```
    public interface Printer{

        public static final int number=10;
        public abstract void print();

    }
```

Implementing an Interface:

- Classes uses **implements** keyword to implement an interface.

Rule:

- The class that implements an interface must override all the abstract methods defined inside that interface, otherwise we need to mark that class as an abstract class.

Example:

ConsolePrinter.java:

```
    public class ConsolePrinter implements Printer {

        public void print() {
            System.out.println("Printing on the console.");
        }
    }
```

FilePrinter.java

```
    public class FilePrinter implements Printer {
        public void print() {
            System.out.println("Printing on the File.");
        }

    }
```

Demo.java

```
    public class Demo{

        public static void main(String args[]) {

            ConsolePrinter cp = new ConsolePrinter();

            //Printer p1 = new Printer(); //CE

            Printer p1 = new ConsolePrinter();
            Printer p2 = new FilePrinter();
            cp.print();
            p1.print();
            p2.print();
        }
    }
```

Note: To an interface variable, we can assign only 2 values.

1. Printer p1 = null;
2. Printer p2 = new ConsolePrinter(); // any of its implemented class objects.

- We are not allowed to create the object of an interface directly.

```
    Printer p3 = new Printer(); //ERROR
```

Here also the rule of the super class reference and subclass object is applicable.

Example: Food Delivery

DeliveryService.java:

```
interface DeliveryService {
    void deliver();
}
```

Zomato.java

```
class Zomato implements DeliveryService {
    public void deliver() {
        System.out.println("Zomato delivers food.");
    }
}
```

Swiggy.java

```
class Swiggy implements DeliveryService {
    public void deliver() {
        System.out.println("Swiggy delivers food.");
    }
}
```

Demo.java:

```
public class Demo {

    public static void main(String[] args) {

        DeliveryService d1 = new Zomato();
        d1.deliver();
```

```
                DeliveryService d2 = new Swiggy();
                d2.deliver();
        }
}
```

Same interface, different behavior.

Interface as method parameter:

- We can also pass an interface as a method parameter.
- To call that method, we can pass either an implementation object or the default value **null**.

Example:

Payment.java:

```
package com.chitkara;
public interface Payment {

        void pay();
}
```

CardPayment.java

```
package com.chitkara;
public class CardPayment implements Payment {

        @Override
        public void pay() {
                System.out.println("Payment done using Card.");
        }
        // Extra method
        public void generateCardReceipt() {
                System.out.println("Card receipt generated.");
        }
}
```

UPIPayment.java:

```
package com.chitkara;
public class UPIPayment implements Payment {

        @Override
        public void pay() {
                System.out.println("Payment done using UPI.");
        }

        // Extra method
        public void showUPIRewards() {
                System.out.println("UPI reward points credited.");
        }
}
```

Demo.java:

```
package com.chitkara;
public class Demo {

        void processPayment(Payment p) {

                // Calling overridden method
                p.pay();

                // Downcasting using instanceof
                if (p instanceof CardPayment) {
                        CardPayment cp = (CardPayment) p;
                        cp.generateCardReceipt();
                } else if (p instanceof UPIPayment) {
```

```
                UPIPayment upi = (UPIPayment) p;
                upi.showUPIRewards();
            }

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

        public static void main(String[] args) {

            Demo d1 = new Demo();

            d1.processPayment(new CardPayment());
            d1.processPayment(new UPIPayment());

        }
    }
```

Interface as a return type of a method:

- A Java method can mention an interface also as a return type.
- If a Java method return type is an interface, then that method can return either any of its implementation objects or it can return null also.

Example:

Hotel.java:

```
package com.chitkara;

public interface Hotel {

    public void chickenBiryani();
    public void masalaDosa();
}
```

TajHotel.java:

```
package com.chitkara;
public class TajHotel implements Hotel {

	@Override
	public void chickenBiryani() {
		System.out.println("ChickenBiryani from TajHotel");
	}

	@Override
	public void masalaDosa() {
		System.out.println("Masala Dosa from TajHotel");
	}

	// specific method of the TajHotel class
	public void paneerMasalaDosa() {
		System.out.println("paneer masala dosa from Taj Hotel");
	}
}
```

RoadSideHotel.java:

```
package com.chitkara;
public class RoadSideHotel implements Hotel {

	@Override
	public void chickenBiryani() {
		System.out.println("ChickenBiryani from RoadSide Hotel");
	}

	@Override
```

```
		public void masalaDosa() {
			System.out.println("ChickenBiryani from RoadSide Hotel");
		}
	}
```

Demo.java:

```
	package com.chitkara;
	public class Demo {

		public Hotel provideFood(int amount) {
			Hotel hotel = null;

			if (amount > 500)
				hotel = new TajHotel();
			else if (amount > 200 && amount <= 500)
				hotel = new RoadSideHotel();

			return hotel;
		}

		public static void main(String[] args) {

			Demo d1 = new Demo();

			Hotel h = d1.provideFood(800);

			if (h != null) {

				h.chickenBiryani();
				h.masalaDosa();

				if (h instanceof TajHotel) {
					TajHotel taj = (TajHotel) h;
					taj.paneerMasalaDosa();
				}
			} else
				System.out.println("Amount should be greater than 200");
		}
	}
```

Multiple Inheritance Using Interface:

- Java does not support multiple inheritance with classes, but with the interface, we can achieve multiple inheritance in Java.
- Java does not support:

```
class C extends A, B // Not allowed
```

- But supports:

```
class C implements A, B
```

- If a class implements multiple interfaces, then that need to override all the abstract methods present inside those interfaces.

Example:

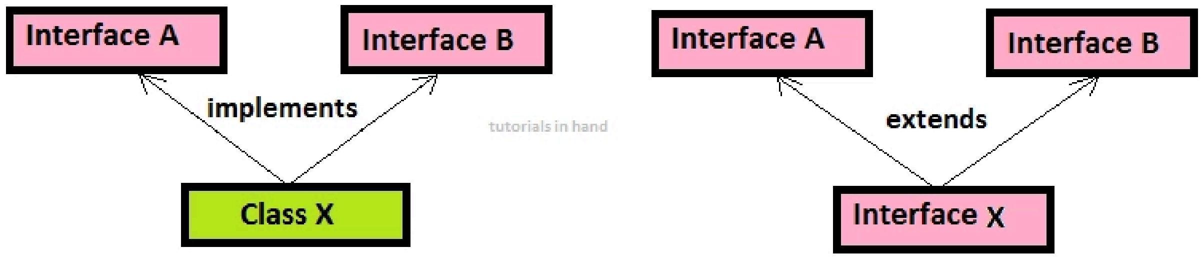
```
interface Camera {
        void clickPhoto();
}

interface MusicPlayer {
        void playMusic();
}

class SmartPhone implements Camera, MusicPlayer {

        public void clickPhoto() {
                System.out.println("Photo clicked");
        }
        public void playMusic() {
                System.out.println("Music playing");
        }
}
```

Demo.java:

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

                SmartPhone phone = new SmartPhone();
                phone.clickPhoto();
                phone.playMusic();

                Camera c = new SmartPhone();
                c.clickPhoto();

                MusicPlayer player = new SmartPhone();
                player.playMusic();

                SmartPhone phone2= (SmartPhone)player;
                phone2.clickPhoto();

        }

}
```

Interface Inheritance:

- A class implements an interface, but one interface extends another interface. Infact one interface can extend multiple interfaces simultaneously.



Multiple inheritance with interface

Example:

interface A: Having 4 abstract methods

interface B: Having 2 abstract methods

interface C: Having 1abstract method

And interface C also extends A, B.

Then, any class that implements the interface C has to override all the 7 methods belongs to all the interfaces, otherwise that class needs to be marked as an abstract class.

Example:

```
interface Printable {
  void print();
}

interface Showable extends Printable {
  void show();
}
```

- The class that implements the Showable interface needs to override print() as well as the show() method; otherwise, we need to mark that class as an **abstract** class.

Note: A Java class can simultaneously extend another class and implement an interface also.

Syntax:

```
class className extends ParentClassName implements InterfaceName {

}
```

Example:

Animal.java:

```
abstract class Animal {

    String name;

    Animal(String name) {
        this.name = name;
    }

    abstract void makeSound();
}
```

Pet.java:

```
interface Pet {

    void play();
}
```

Dog.java:

```
class Dog extends Animal implements Pet {

    Dog(String name) {
        super(name);
    }

    @Override
    void makeSound() {
        System.out.println("Bark");
    }

    @Override
    public void play() {
        System.out.println("Dog playing");
    }
}
```

Demo.java:

```
public class Demo{

        public static void main(String[] args) {


                Dog d1= new Dog("Tommy");
                d1.makeSound();
                d1.play();

                Pet p = new Dog("tommy");
                p.play();


                Animal a = new Dog("tommy");
                a.makeSound();

        }
}
```

New Features Added in Interfaces in JDK 8:

1. The default method inside an interface:

- Before **Java 8**, interfaces could only contain:
    - Abstract methods
    - Constants
- They **could not contain method implementations**.

Problem:

- If you added a new method to an existing interface, **all implementing classes would break** because they must implement that new method.
- To solve this issue, Java 8 introduced **default methods**.

What is a Default Method?

- A **default method** is a method inside an interface that:
  - Has a method body (implementation)
  - Uses the keyword `default`
  - Is automatically inherited by implementing classes
  - Can be optionally overridden

Analogy:

- Imagine a **Vehicle interface** used by many companies.
- Suddenly, the government says:

  "All vehicles must have a GPS system."
- Now, all existing vehicle classes (Car, Bike, Truck) will break if we add a new method like `startGPS()`.
- Instead, we provide a **default GPS implementation** inside the interface.
- Old vehicles continue to work without any changes.

Syntax:

```
interface InterfaceName {

    void abstractMethod();

    default void newMethod() {
```

```
                // implementation
        }
}
```

Example 1 – Without Overriding Default Method

Vehicle.java:

```
interface Vehicle {

  void start();

  default void startGPS() {
      System.out.println("Starting default GPS system...");
  }

}
```

Car.java

```
class Car implements Vehicle {
  public void start() {
      System.out.println("Car is starting...");
  }
}
```

Demo.java

```
class Demo{

  public static void main(String[] args) {

      Car c = new Car();
      c.start();
      c.startGPS();   // inherited default method
```

```
    }
}
```

Output:

```
Car is starting…

Starting default GPS system...
```

Here:

- Car did NOT implement `startGPS()`
- Still it works
- It inherited the default implementation.

Example 2 – Overriding Default Method

- Implementation classes can override the default method if needed.

Bike.java

```
class Bike implements Vehicle {

    public void start() {
        System.out.println("Bike is starting...");
    }

    // Overriding default method
    public void startGPS() {
        System.out.println("Bike GPS system started.");
    }
}
```

Demo.java

```
class Demo{

  public static void main(String[] args) {

      Vehicle v1 = new Car();
      Vehicle v2 = new Bike();

      v1.startGPS();  // Default version
      v2.startGPS();  // Overridden version

  }
}
```

Output:

```
Starting default GPS system…

Bike GPS system started.
```

Important Rules

1. Default methods are **public** by default.
2. A class can override them.
3. If a class implements two interfaces with the same default method, the class must override it to remove ambiguity.

Diamond Problem Example:

```
interface A {
  default void show() {
      System.out.println("A");
  }
}


interface B {

  default void show() {
      System.out.println("B");
  }
}

class Demo implements A, B {

  public void show() {
      System.out.println("Resolving conflict");
  }

}
```

- Demo class must override the show() method to resolve the conflict.

2. The static method inside an interface:

A **static method inside an interface**:

- Has a body (implementation)
- Belongs to the interface itself
- Is called using the **interface name**
- Is **NOT inherited** by implementing classes

Analogy:

Think of an interface as a **company's rule book**.

- Abstract methods → Rules employees must follow.
- Default methods → Standard behavior provided
- Static methods → Utility functions of the company

For example:

Company Rule Book may have:

- Work policy (abstract method)
- Default leave policy (default method)
- Company helpline number (static method)

Helpline belongs to company, not employees.

Syntax:

```
interface InterfaceName {

  static void methodName() {
     // implementation
  }

}
```

Example:

Bank.java

```
interface Bank {

        void withdraw();

        static void bankRules() {
                System.out.println("Bank timing: 9 AM to 5 PM");
        }

}
```

SBI.java:

```
class SBI implements Bank {

        public void withdraw() {
                System.out.println("Money withdrawn from SBI");
        }
}
```

Demo.java:

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

     SBI s = new SBI();
     s.withdraw();
     Bank.bankRules();  // static method call
  }
}
```

Purpose of the static methods in the interface?

1. **Utility Methods**

- Common helper methods related to the interface.

2. **Code Organization**
   - Keep related logic inside the interface.
3. **Avoid Separate Utility Class**
   - No need to create an extra helper class.

Difference Between Default and Static Method:

Default Method	**Static Method**
Belongs to the object	Belongs to the interface
Can be overridden	Cannot be overridden
Inherited	Not inherited
Called using an object	Called using the interface name

Private Methods in Interface (Java 9 Feature):

Before **Java 9**, interfaces could have:

- Abstract methods
- Default methods (Java 8)
- Static methods (Java 8)

- But sometimes, **default and static methods need common internal logic**.
- To avoid repeating code inside the interface, **Java 9 introduced private methods inside interfaces**.

Why Private Methods Were Added?

- Suppose you have multiple default methods inside an interface, and both use the same logic.

  Without private methods: You must duplicate code.

  With private methods: You write helper logic once and reuse it.

Analogy:

Think of an interface as a **company rulebook**.

- **Abstract methods:** Rules employees must follow
- **Default methods:** Standard company procedures
- **Static methods:** Company utilities
- **Private methods:** Internal secret procedures used only inside company

Employees (implementing classes) cannot access private methods.

Syntax

```
interface InterfaceName {

  private void helperMethod() {
      // code
  }

  default void method1() {
      helperMethod();
  }
}
```

Example:

Payment.java

```
interface Payment {
        default void makePayment() {
            validatePayment();
            System.out.println("Payment processed.");
        }

        private void validatePayment() {
            System.out.println("Validating payment...");
        }
    }
```

CreditCard.java:

```
    class CreditCard implements Payment {

    }
```

Demo.java:

```
    class Demo {

        public static void main(String[] args) {

            Payment p = new CreditCard();
            p.makePayment();
            //p.validatePayment(); //ERROR
        }
    }
```

Marker (Tagged) Interface in Java:

- A **Marker Interface** (also called a **Tagged Interface**) is an interface that:
    - Has **no methods**
    - Has **no variables**
    - Is completely empty

    - Is used to “mark” a class

- It provides **special information to the JVM or compiler**.

Simple Definition

- A marker interface is an empty interface used to indicate that a class has a special property.

Marker interfaces are used to:

- Provide special instructions to JVM
- Enable certain features
- Indicate the capability of a class.

Analogy:

- Imagine a college ID card system.
- Some students have a **"Sports Player" badge**.
- That badge does not contain any instructions.
- But it tells the college:

    "This student is allowed to use sports facilities."

Similarly:

Marker interface = Special badge
Class = Student
JVM = College authority

Example 1: java.io.Serializable

- One of the most common marker interfaces:
- When a class implements `Serializable`, it tells the JVM:

  "Objects of this class can be converted into a byte stream."

Example 2: java.lang.Cloneable

- If a class implements Cloneable, it tells the JVM:

  "Objects of this class can be cloned."

The JVM checks:

```
if(object instanceof Serializable)
```

If true: Allow serialization.
If false: Throw exception.

Creating Our Own Marker Interface:

PremiumUser.java:

```
interface PremiumUser {
}
```

Customer.java:

```
class Customer implements PremiumUser {
}
```

Now we can check:

```
Customer c1 = new Customer();

if(c1 instanceof PremiumUser) {
  System.out.println("Give special discount");
}
```

Difference between abstract class and interface:

- Abstract class and interface are both used to achieve abstraction, where we can declare the abstract methods.
- Abstract class and interface both can't be instantiated.
- But many differences between an abstract class and an interface are given below.

Abstract class	**Interface**
Using an abstract class, we achieve partial abstraction	Using an interface, we achieve 100% abstraction
An Abstract class can have abstract and non-abstract methods.	An Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
An Abstract class doesn't support multiple inheritance.	An Interface supports multiple inheritance.
An Abstract class can have final, non-final, static, and non-static variables.	An Interface has only static and final variables.
An Abstract class can provide the implementation of an interface.	An Interface can't provide the implementation of an abstract class
An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.