protected Members in Inheritance

```
class Animal {
 protected String name;
 protected void display() {
  System.out.println("I am an animal.");
 }
}
class Dog extends Animal {
 public void getInfo() {
  System.out.println("My name is " + name);
 }
}
class Main {
 public static void main(String[] args) {
  Dog labrador = new Dog();
  labrador.name = “Pinky";
  labrador.display();

  labrador.getInfo();
 }
}
```

Why use inheritance?

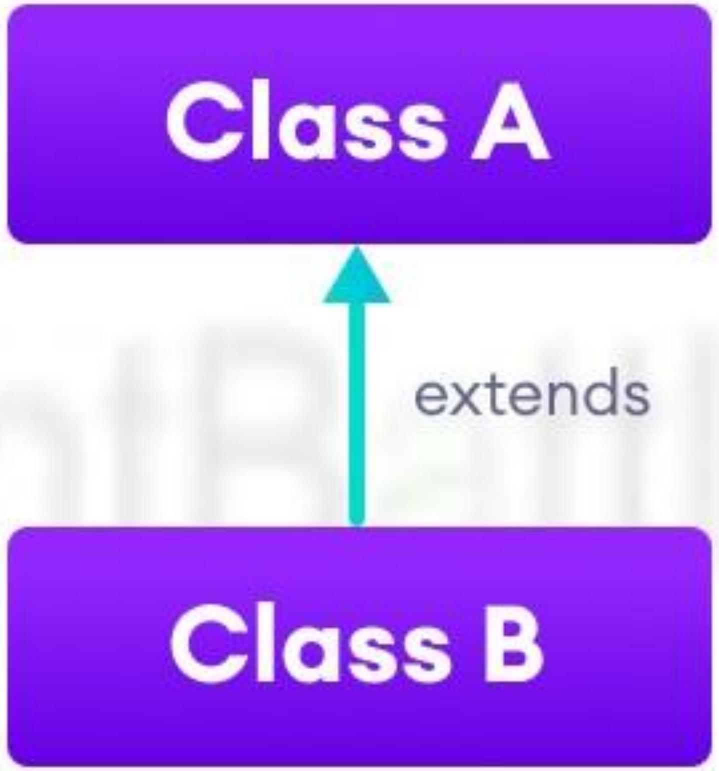
•The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.

•Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

Types of inheritance

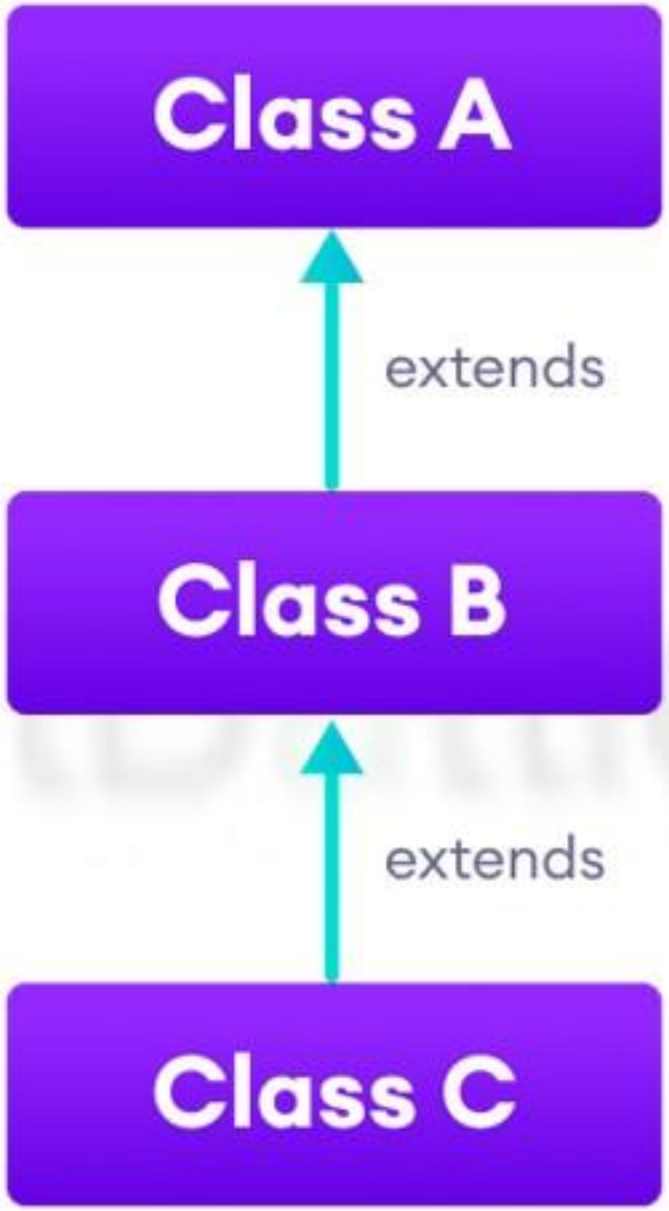
There are five types of inheritance.

1. Single Inheritance
In single inheritance, a single subclass extends from a single superclass.



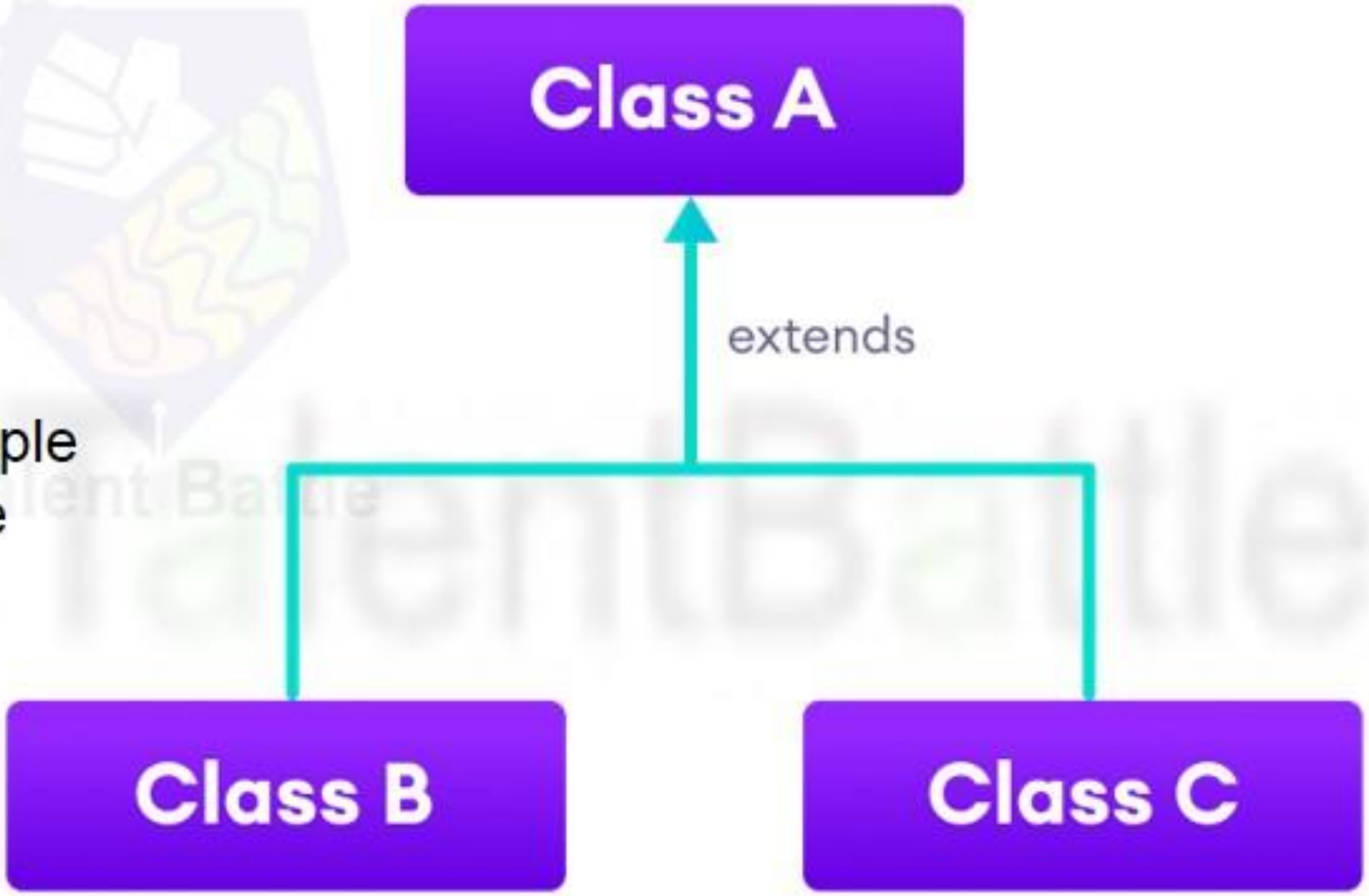
2. Multilevel Inheritance

In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class.



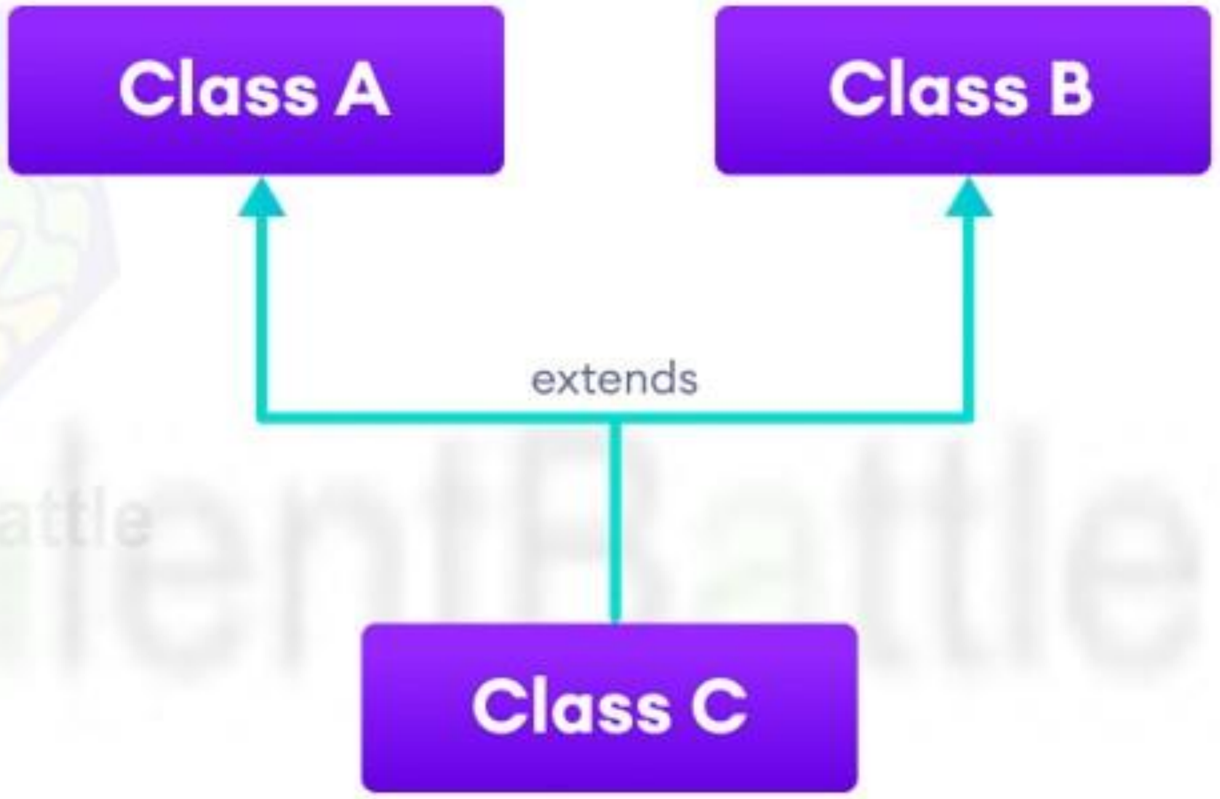
3. Hierarchical Inheritance

In hierarchical inheritance, multiple subclasses extend from a single superclass.



4. Multiple Inheritance

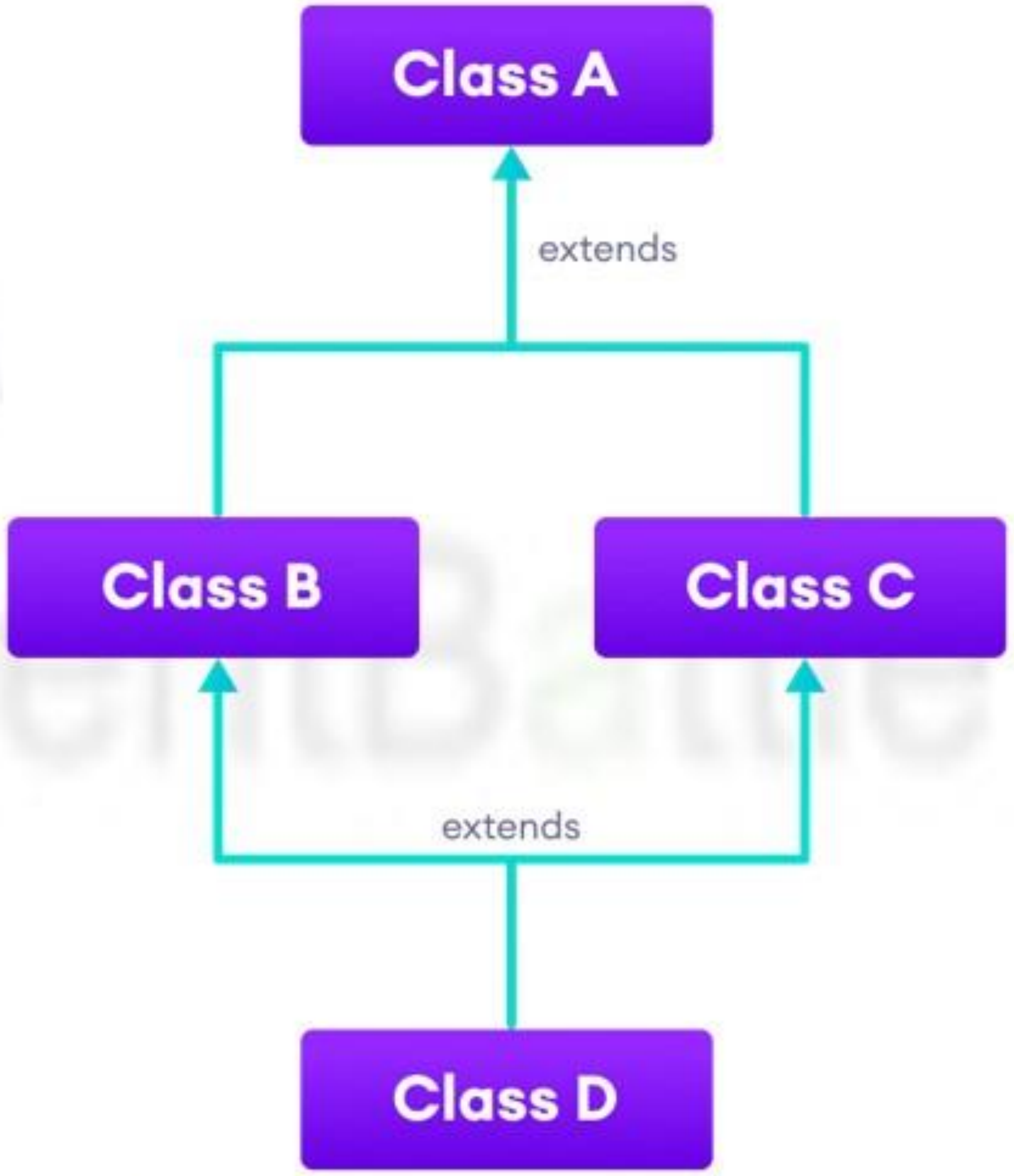
In multiple inheritance, a single subclass extends from multiple superclasses.



Note: Java doesn't support multiple inheritance.

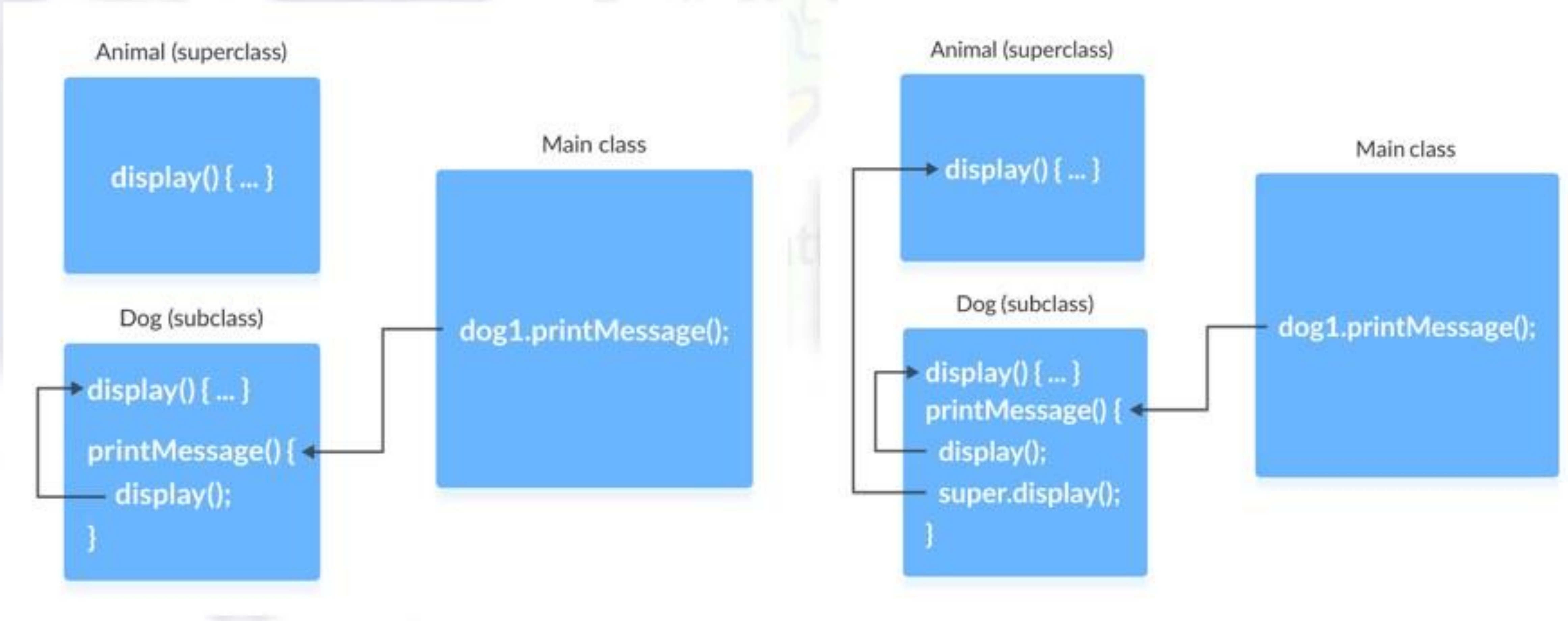
5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance.



Java super

The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).



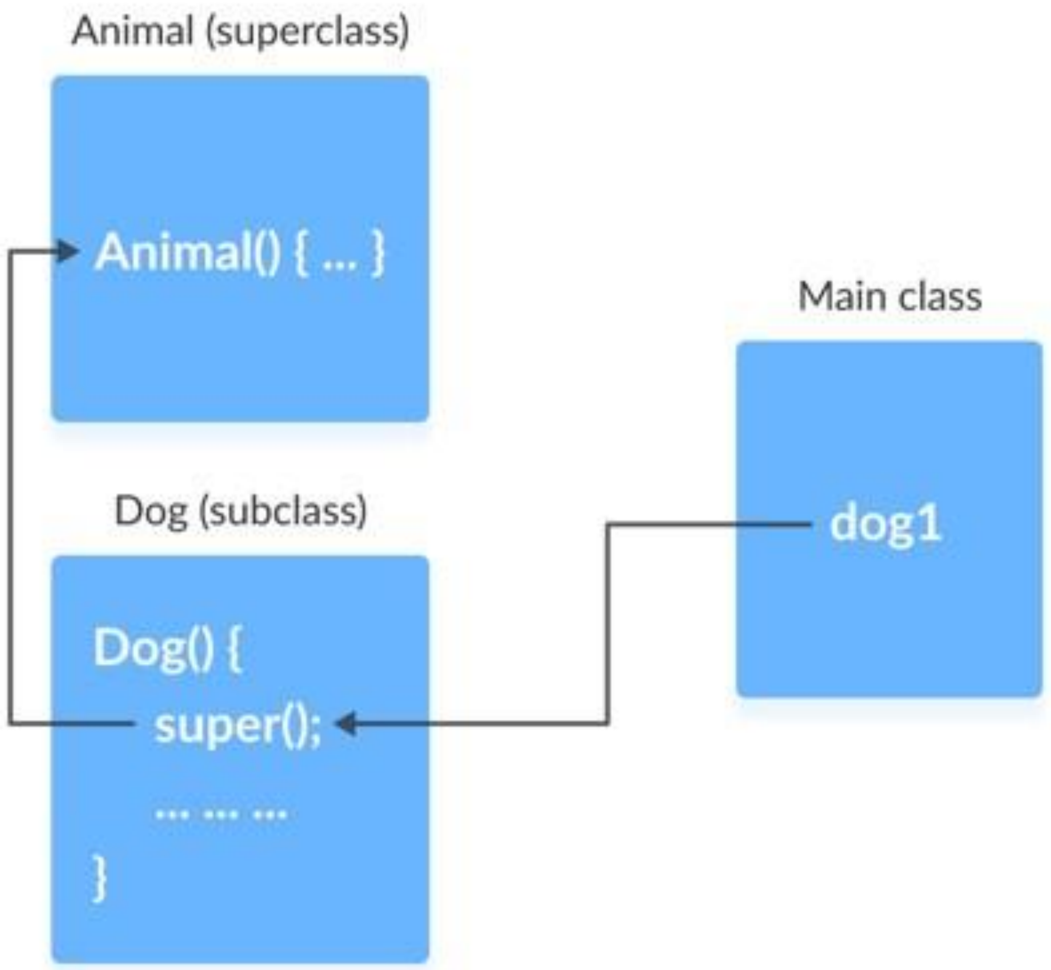
Use of super() to access superclass constructor

As we know, when an object of a class is created, its default constructor is automatically called.

To explicitly call the superclass constructor from the subclass constructor, we use super(). It's a special form of the super keyword.

super() can be used only inside the subclass constructor and must be the first statement.

```
class Animal {
 // default or no-arg constructor of class Animal
 Animal() {
  System.out.println("I am an animal");
 }
}
class Dog extends Animal {
 // default or no-arg constructor of class Dog
 Dog() {
  // calling default constructor of the superclass
  super();
  System.out.println("I am a dog");
 }
}
class Main {
 public static void main(String[] args) {
  Dog dog1 = new Dog();
 }
}
```



Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
 // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods.
For example,

```
abstract class Language {

 // abstract method
 abstract void method1();

 // regular method
 void method2() {
  System.out.println("This is regular method");
 }
}
```

Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. For example,

```
abstract void display();
```

Here, display() is an abstract method. The body of display() is replaced by ;.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {

 // abstract method
 abstract void method1();
}
```

```
abstract class Language {
 // method of abstract class
 public void display() {
  System.out.println("This is Java Programming");
 }
}
class Main extends Language {
 public static void main(String[] args) {

  // create an object of Main
  Main obj = new Main();

  // access method of abstract class
  // using object of Main class
  obj.display();
 }
}
```

Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

```
abstract class Animal {
 abstract void makeSound();
 public void eat() {
  System.out.println("I can eat.");
 }
}
class Dog extends Animal {
 // provide implementation of abstract method
 public void makeSound() {
  System.out.println("Bark bark");
 }
}
class Main {
 public static void main(String[] args) {
  // create an object of Dog class
  Dog d1 = new Dog();
  d1.makeSound();
  d1.eat();
 }
}
```

Java Abstraction

The major use of abstract classes and methods is to achieve abstraction in Java.

Abstraction is an important concept of object-oriented programming that allows us to hide unnecessary details and only show the needed information.

This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

A practical example of abstraction can be motorbike brakes. We know what brake does. When we apply the brake, the motorbike will stop. However, the working of the brake is kept hidden from us.

The major advantage of hiding the working of the brake is that now the manufacturer can implement brake differently for different motorbikes, however, what brake does will be the same.

```
abstract class Animal {
 abstract void makeSound();
}
class Dog extends Animal {
 // implementation of abstract method
 public void makeSound() {
  System.out.println("Bark bark.");
 }
}
class Cat extends Animal {
 // implementation of abstract method
 public void makeSound() {
  System.out.println("Meows ");
 }
}
class Main {
 public static void main(String[] args) {
  Dog d1 = new Dog();
  d1.makeSound();
  Cat c1 = new Cat();
  c1.makeSound();
 }
}
```

Key Points to Remember

- We use the abstract keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class.

Java Interface

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the interface keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

Language is an interface.
It includes abstract methods: getType() and getVersion().

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the **implements** keyword to implement an interface.

```
interface Polygon {
 void getArea(int length, int breadth);
}
// implement the Polygon interface
class Rectangle implements Polygon {
 // implementation of abstract method
 public void getArea(int length, int breadth) {
  System.out.println("The area of the rectangle is " + (length * breadth));
 }
}
class Main {
 public static void main(String[] args) {
  Rectangle r1 = new Rectangle();
  r1.getArea(5, 6);
 }
}
```

Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A {
 // members of A
}

interface B {
 // members of B
}

class C implements A, B {
 // abstract members of A
 // abstract members of B
}
```

Extending an Interface

Similar to classes, interfaces can extend other interfaces. The **extends** keyword is used for extending interfaces. For example,

```
interface Line {
 // members of Line interface
}

// extending interface
interface Polygon extends Line {
 // members of Polygon interface
 // members of Line interface
}
```

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

```
interface A {
   ...
}
interface B {
   ...
}

interface C extends A, B {
   ...
}
```

Advantages of Interface in Java

- Similar to abstract classes, interfaces help us to achieve **abstraction in Java**.
- Interfaces **provide specifications** that a class (which implements it) must follow.

Interfaces are also used to achieve multiple inheritance in Java.

```
interface Line {
...
}
interface Polygon {
...
}
class Rectangle implements Line, Polygon {
...
}
```

Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

Note: All the methods inside an interface are implicitly public and all fields are implicitly public static final.

default methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the default keyword. For example,

```
public default void getSides() {
  // body of getSides()
}
```

Example: Default Method in Java Interface

```
interface Polygon {
 void getArea();
 // default method
 default void getSides() {
  System.out.println("I can get sides of a polygon.");
 }
}
// implements the interface
class Rectangle implements Polygon {
 public void getArea() {
  int length = 6;
  int breadth = 5;
  int area = length * breadth;
  System.out.println("The area of the rectangle is " + area);
 }
 // overrides the getSides()
 public void getSides() {
  System.out.println("I have 4 sides.");
 }
}
// implements the interface
class Square implements Polygon {
 public void getArea() {
  int length = 5;
  int area = length * length;
  System.out.println("The area of the square is " + area);
 }
}
class Main {
 public static void main(String[] args) {
  // create an object of Rectangle
  Rectangle r1 = new Rectangle();
  r1.getArea();
  r1.getSides();
  // create an object of Square
  Square s1 = new Square();
  s1.getArea();
  s1.getSides();
 }
}
```

Java Polymorphism

Polymorphism is an important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

```
class Polygon {
 // method to render a shape
 public void render() {
  System.out.println("Rendering Polygon...");
 }
}
class Square extends Polygon {
 // renders Square
 public void render() {
  System.out.println("Rendering Square...");
 }
}
class Circle extends Polygon {
 // renders circle
 public void render() {
  System.out.println("Rendering Circle...");
 }
}
class Main {
 public static void main(String[] args) {
  // create an object of Square
  Square s1 = new Square();
  s1.render();
  // create an object of Circle
  Circle c1 = new Circle();
  c1.render();
 }
}
```

Why Polymorphism?

Polymorphism allows us to create consistent code. In the previous example, we can also create different methods: renderSquare() and renderCircle() to render Square and Circle, respectively.

This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent.

To solve this, polymorphism in Java allows us to create a single method render() that will behave differently for different shapes.

Note: The print() method is also an example of polymorphism. It is used to print values of different types like char, int, string, etc.