Java Access Modifiers

What are Access Modifiers?

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

```
class Animal {
   public void method1() {...}

  private void method2() {...}
}
```

In the above example, we have declared 2 methods: method1() and method2(). Here,

method1 is public - This means it can be accessed by other classes.
method2 is private - This means it can not be accessed by other classes.
Note the keyword public and private. These are access modifiers in Java. They are also known as visibility modifiers.

Note: You cannot set the access modifier of getters methods.

Types of Access Modifier

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

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

Default Access Modifier

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

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

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

Private Access Modifier

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

```
class Data {
  // private variable
  private String name;
}
public class Main {
  public static void main(String[] main){
    // create an object of Data
    Data d = new Data();
    // access private variable and field from another class
    d.name = "Programming";
  }
}
```

In the above example, we have declared a private variable named name and a private method named display().

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

```
class Data {
  private String name;
  // getter method
  public String getName() {
    return this.name;
  }
  // setter method
  public void setName(String name) {
    this.name= name;
  }
}
public class Main {
  public static void main(String[] main){
    Data d = new Data();
    // access the private variable using the getter and setter
    d.setName("Programming");
    System.out.println(d.getName());
  }
}
```

Note: We cannot declare classes and interfaces private in Java.

Protected Access Modifier

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

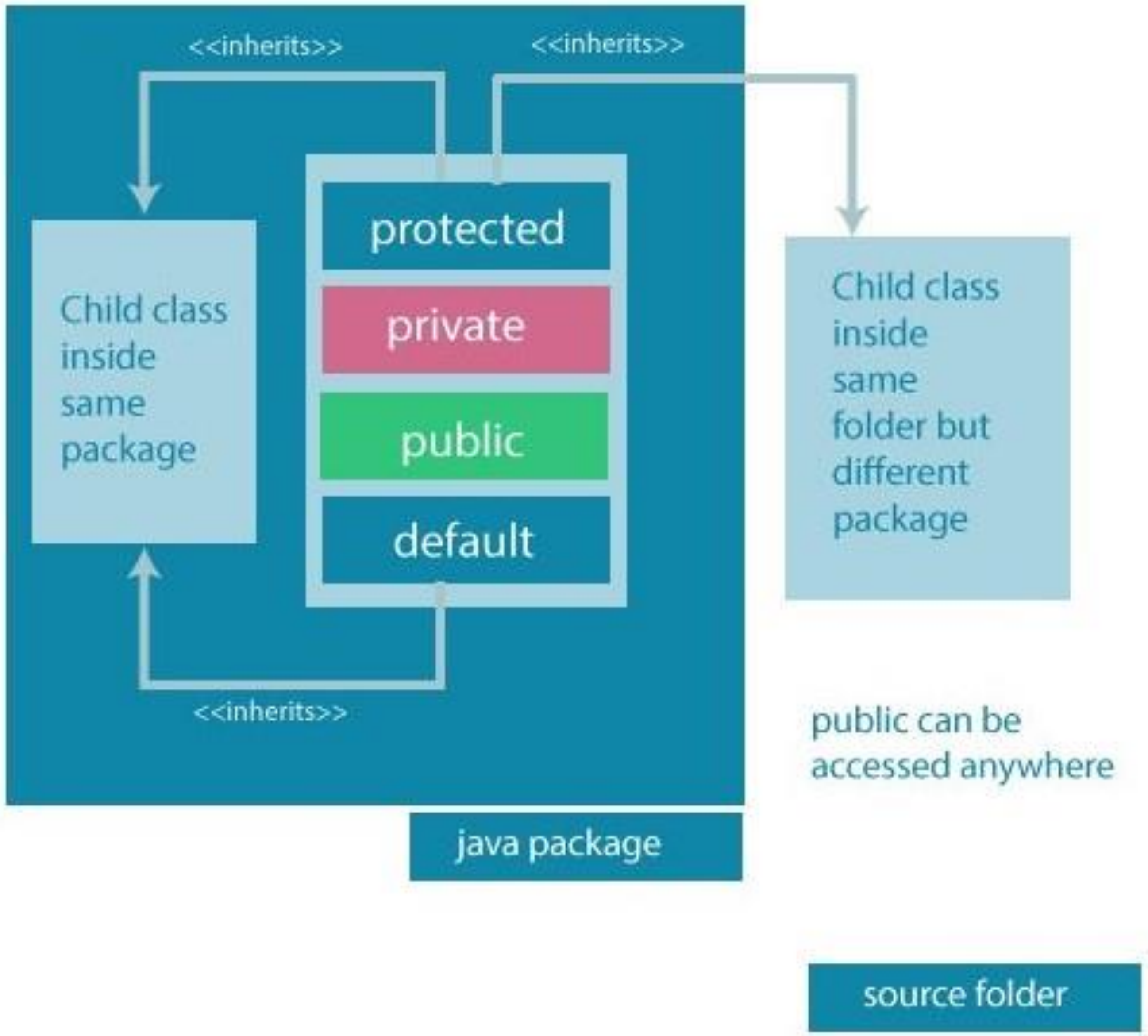
```
class Animal {
   // protected method
   protected void display() {
      System.out.println("I am an animal");
   }
}
class Dog extends Animal {
   public static void main(String[] args) {
      // create an object of Dog class
      Dog dog = new Dog();
       // access protected method
      dog.display();
   }
}
```

Note: We cannot declare classes or interfaces protected in Java.

Public Access Modifier

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

```
public class Animal {
   // public variable
   public int legCount;
   // public method
   public void display() {
      System.out.println("I am an animal.");
      System.out.println("I have " + legCount + " legs.");
   }
}
// Main.java
public class Main {
   public static void main( String[] args ) {
      // accessing the public class
      Animal animal = new Animal();
      // accessing the public variable
      animal.legCount = 4;
      // accessing the public method
      animal.display();
   }
}
```



<<inherits>>
<<inherits>>
protected
private
public
default
Child class inside same package
Child class inside same folder but different package
<<inherits>>
public can be accessed anywhere
java package
source folder

Access modifiers are mainly used for encapsulation. It can help us to control what part of a program can access the members of a class. So that misuse of data can be prevented.

Java this Keyword

this Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor. For example,

```
class Main {
    int instVar;
    Main(int instVar){
        this.instVar = instVar;
        System.out.println("this reference = " + this);
    }
    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("object reference = " + obj);
    }
}
```

this is nothing but the reference to the current object

Using this for Ambiguity Variable Names

In Java, it is not allowed to declare two or more variables having the same name inside a scope (class scope or method scope). However, instance variables and parameters may have the same name. For example,

```
class MyClass {
    // instance variable
    int age;

    // parameter
    MyClass(int age){
        age = age;
    }
}
```

In the above program, the instance variable and the parameter have the same name: age. Here, the Java compiler is confused due to name ambiguity.

First, let's see an example without using this keyword:

```
class Main {

    int age;
    Main(int age){
        age = age;
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("obj.age = " + obj.age);
    }
}
```

Output:

```
mc.age = 0
```

Now, let's rewrite the code using **this** keyword.

```
class Main {

  int age;
  Main(int age){
    this.age = age;
  }

  public static void main(String[] args) {
    Main obj = new Main(8);
    System.out.println("obj.age = " + obj.age);
  }
}
```

Output:

```
obj.age = 8
```

Now, we are getting the expected output. It is because when the constructor is called, this inside the constructor is replaced by the object obj that has called the constructor. Hence the age variable is assigned value 8.

this with Getters and Setters

```
class Main {
  String name;

  // setter method
  void setName( String name ) {
    this.name = name;
  }
  // getter method
  String getName(){
    return this.name;
  }
  public static void main( String[] args ) {
    Main obj = new Main();
    // calling the setter and the getter method
    obj.setName("Toshiba");
    System.out.println("obj.name: "+obj.getName());
  }
}
```

Here, we have used this keyword:

- to assign value inside the setter method
- to access value inside the getter method

Using this in Constructor Overloading

While working with constructor overloading, we might have to invoke one constructor from another constructor.

In such a case, we cannot call the constructor explicitly. Instead, we have to use this keyword.

Here, we use a different form of this keyword. That is, **this()**.

Java final keyword

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

In Java, we cannot change the value of a final variable. For example,

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

  // create a final variable
  final int AGE = 32;

  // try to change the final variable
  AGE = 45;
  System.out.println("Age: " + AGE);
 }
}
```

Note: It is recommended to use uppercase to declare final variables in Java

Java final Method

In Java, the final method cannot be overridden by the child class. For example,

```
class FinalDemo {
   // create a final method
   public final void display() {
     System.out.println("This is a final method.");
   }
}
class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }
  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

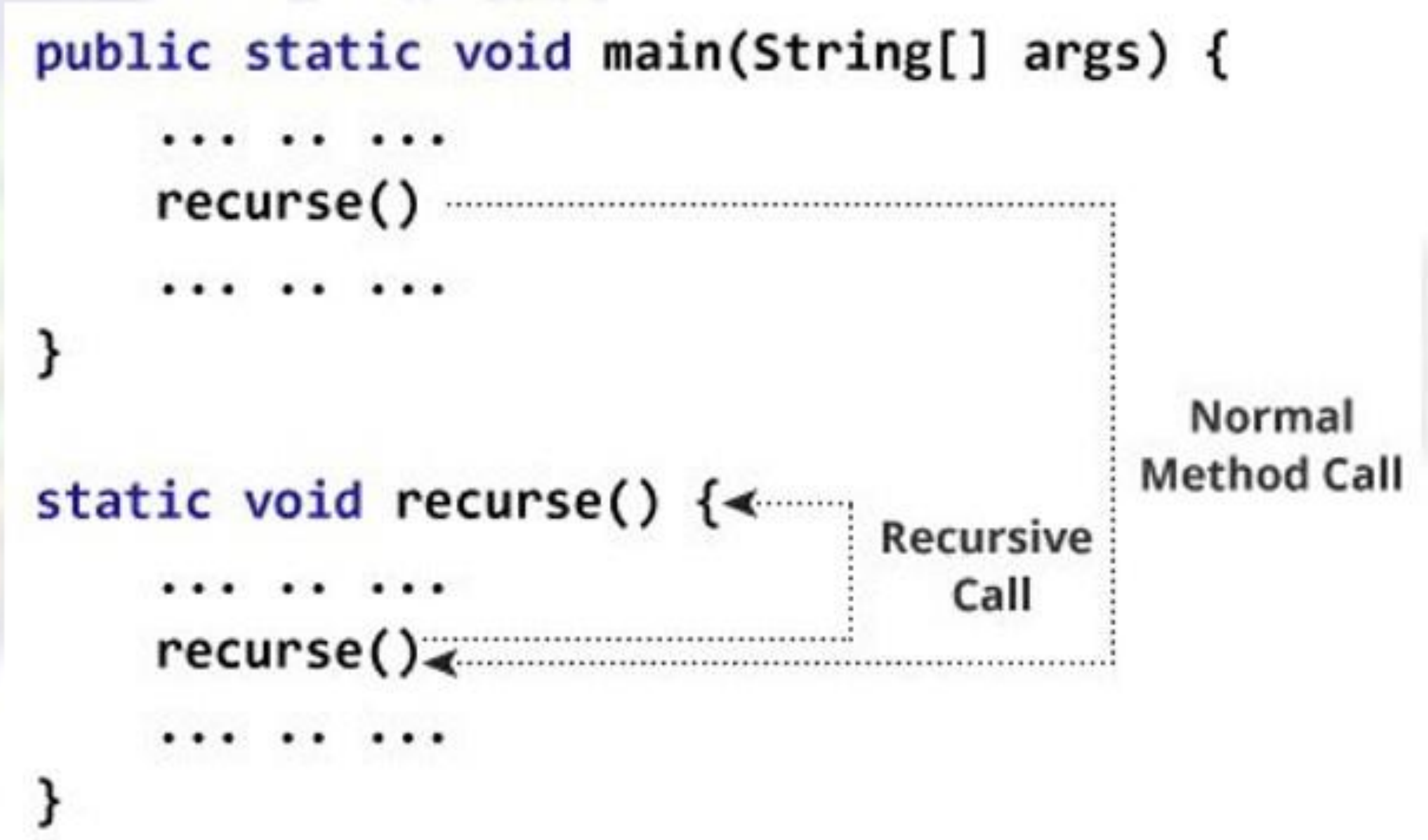
Java final Class

In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass {
 public void display() {
  System.out.println("This is a final method.");
 }
}
// try to extend the final class
class Main extends FinalClass {
 public  void display() {
  System.out.println("The final method is overridden.");
 }
 public static void main(String[] args) {
  Main obj = new Main();
  obj.display();
 }
}
```

Java Recursion

In Java, a method that calls itself is known as a recursive method. And, this process is known as recursion.



```
public static void main(String[] args) {
    ... .. ...
    recurse()
    ... .. ...
}

static void recurse() {
    ... .. ...
    recurse()
    ... .. ...
}
```

Example: Factorial of a Number Using Recursion

```
class Factorial {

    static int factorial( int n ) {
        if (n != 0)  // termination condition
            return n * factorial(n-1); // recursive call
        else
            return 1;
    }

    public static void main(String[] args) {
        int number = 4, result;
        result = factorial(number);
        System.out.println(number + " factorial = " + result);
    }
}
```

Java instanceof Operator

The **instanceof** operator in Java is used to check whether an object is an instance of a particular class or not.

Its syntax is

objectName instanceOf className;

Here, if objectName is an instance of className, the operator returns true. Otherwise, it returns false.

```
class Main {
  public static void main(String[] args) {
    // create a variable of string type
   String name = "Programming";
  // checks if name is instance of String
    boolean result1 = name instanceof String;
    System.out.println("name is an instance of String: " + result1);
    // create an object of Main
    Main obj = new Main();
    // checks if obj is an instance of Main
    boolean result2 = obj instanceof Main;
    System.out.println("obj is an instance of Main: " + result2);
  }
}
```

Java Inheritance

Inheritance is one of the key features of OOP that allows us to create a new class from an existing class.

The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class).

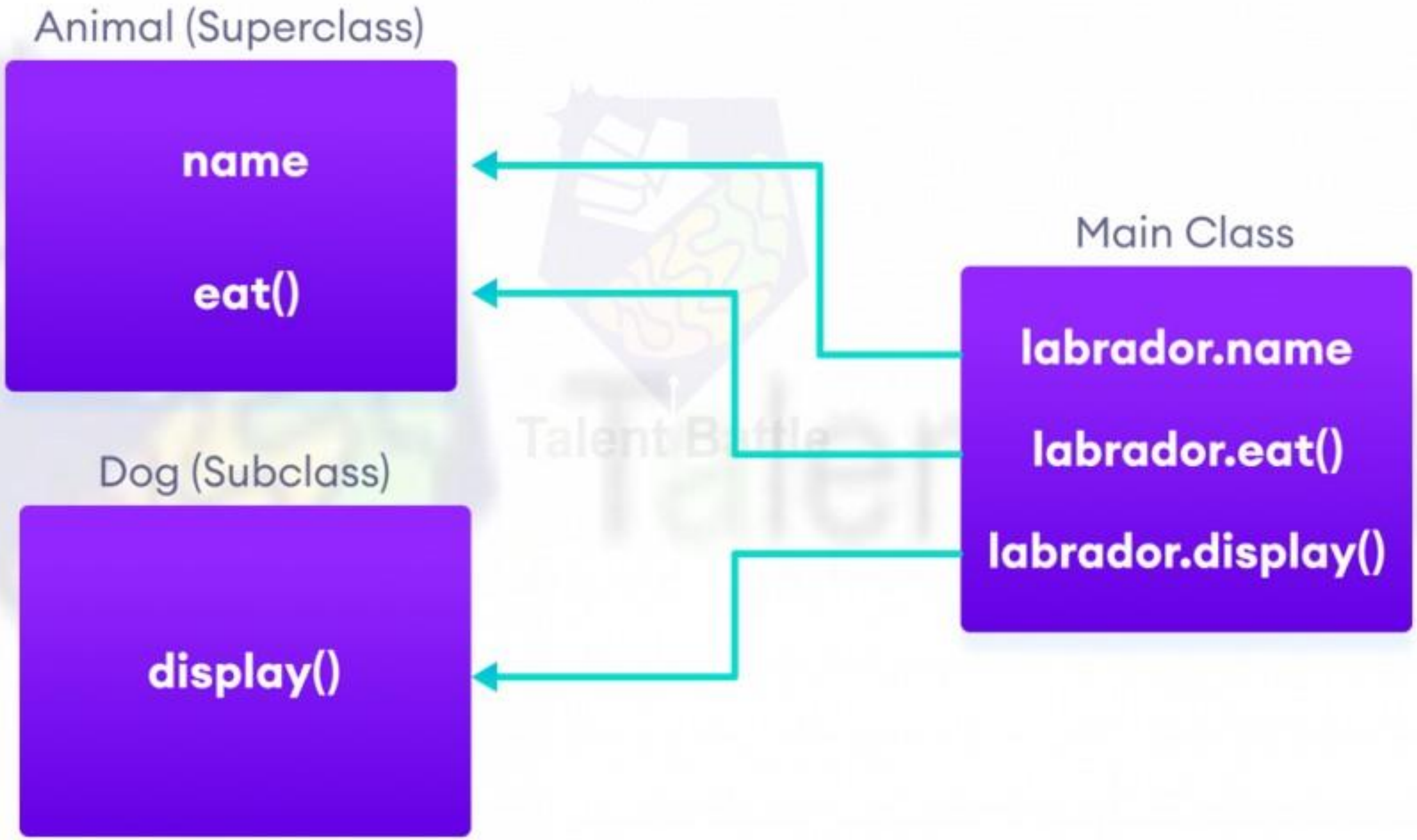
The **extends** keyword is used to perform inheritance in Java.

```
class Animal {
 // methods and fields
}

// use of extends keyword
// to perform inheritance
class Dog extends Animal {

 // methods and fields of Animal
 // methods and fields of Dog
}
```

In the above example, the Dog class is created by inheriting the methods and fields from the Animal class.

Here, Dog is the subclass and Animal is the superclass.

```
class Animal {
 // field and method of the parent class
 String name;
 public void eat() {
  System.out.println("I can eat");
 }
}
class Dog extends Animal {
  public void display() {
  System.out.println("My name is " + name);
 }
}
class Main {
 public static void main(String[] args) {
    Dog labrador = new Dog();
    labrador.name = "Tommy";
  labrador.display();
    labrador.eat();
 }
}
```


Animal (Superclass)
name
eat()
Dog (Subclass)
display()
Main Class
labrador.name
labrador.eat()
labrador.display()

is-a relationship

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

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

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Method Overriding in Java Inheritance

```
class Animal {
 public void eat() {
  System.out.println("I can eat");
 }
}
class Dog extends Animal {
 // overriding the eat() method
 @Override
 public void eat() {
  System.out.println("I eat dog food");
 }
 // new method in subclass
 public void bark() {
  System.out.println("I can bark");
 }
}
class Main {
 public static void main(String[] args) {
  // create an object of the subclass
  Dog labrador = new Dog();
  // call the eat() method
  labrador.eat();
  labrador.bark();
 }
}
```

super Keyword in Java Inheritance

Previously we saw that the same method in the subclass overrides the method in superclass.

In such a situation, the super keyword is used to call the method of the parent class from the method of the child class.

```
class Animal {
  public void eat() {
    System.out.println("I can eat");
  }
}
class Dog extends Animal {
  // overriding the eat() method
  @Override
  public void eat() {
    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
  }
  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}
class Main {
  public static void main(String[] args) {
    Dog labrador = new Dog();
    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```