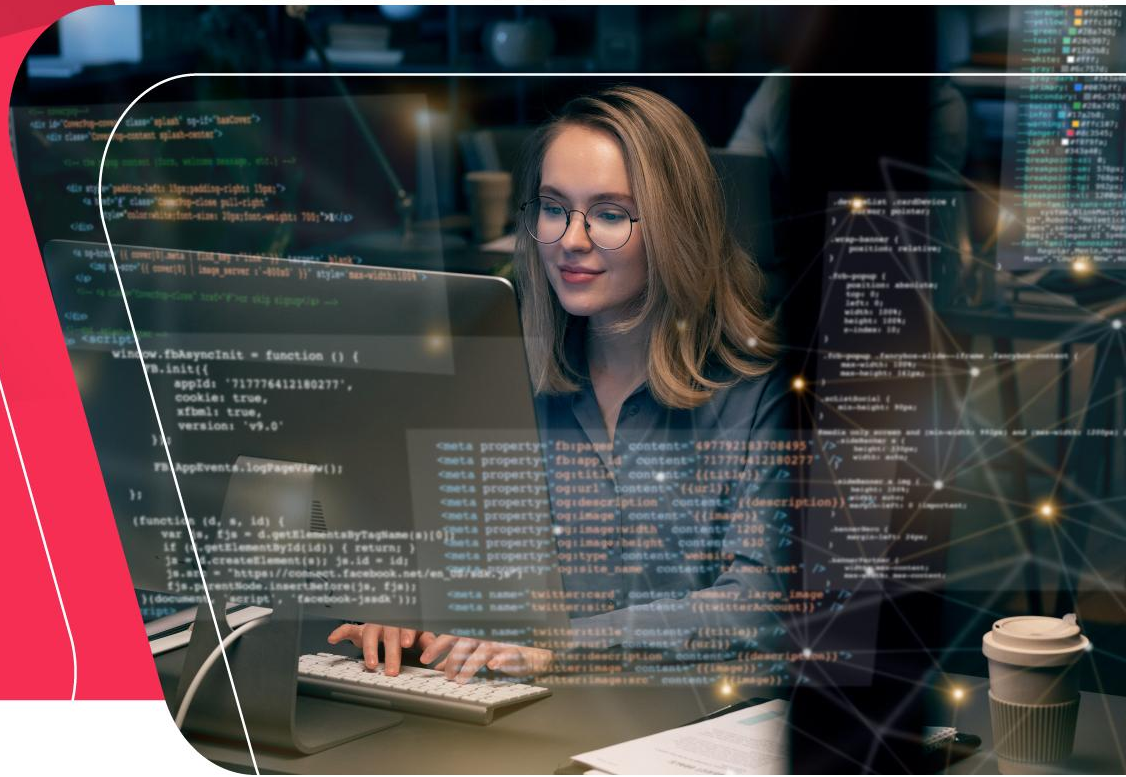
TIA-Academy

Java Bootcamp

Day 16



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- Writing Code using Notepad++ (For Windows) or Vim (For Linux and Mac)
- Compiling with Command Prompt (For Windows) or Terminal (For Linux and Mac)
- ***Forbidden of using IDE Based Development Tools as of Intellij IDEA or Eclipse***

TIA-Academy

Java Inheritance

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).

Java Inheritance

- The **extends** keyword is used to perform inheritance in Java. For example,

```
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.

Java Inheritance

- Example 1: Java Inheritance.
- Output:

```
My name is Rohu
I can eat
```

```
class Animal {
  // field and method of the parent class
  String name;
  public void eat() {
    System.out.println("I can eat");
  }
}

// inherit from Animal
class Dog extends Animal {
  // new method in subclass
  public void display() {
    System.out.println("My name is " + name);
  }
}

class Main {
  public static void main(String[] args) {
    // create an object of the subclass
    Dog labrador = new Dog();

    // access field of superclass
    labrador.name = "Rohu";
    labrador.display();

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

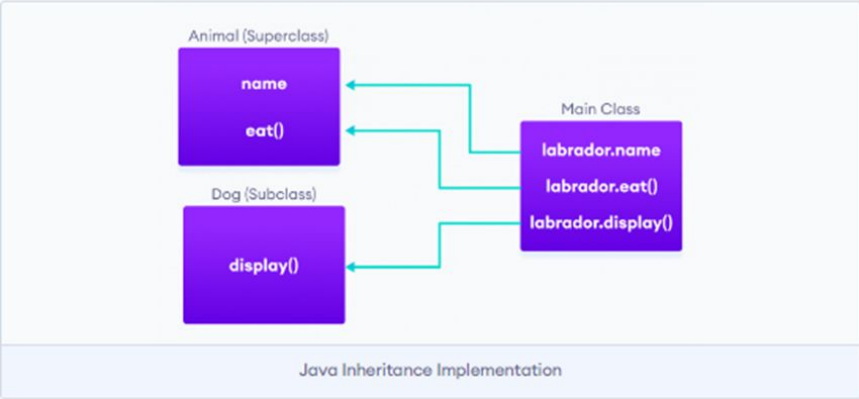
Java Inheritance

- In the above example, **we have derived a subclass Dog** from **superclass Animal**. Notice the statements,

```
labrador.name = "Rohu";

labrador.eat();
```

Java Inheritance

- Here, **labrador is an object of Dog**. However, **name and eat() are the members of the Animal class**.
- Since **Dog inherits** the field and method from Animal, **we are able to access the field and method using the object of the Dog**.



Java Inheritance Implementation

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

- In **Example 1**, we see the object of the subclass can access the method of the superclass.
- **However, if the same method is present in both the superclass and subclass, what will happen?**
- In this case, the method in the subclass overrides the method in the superclass. This concept is known as **method overriding** in Java.

Method Overriding in Java Inheritance

- Example 2: Method overriding in Java Inheritance

```
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
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");
  }
}
```

Method Overriding in Java Inheritance

```
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();
  }
}
```

- Output :

```
I eat dog food
I can bark
```

Method Overriding in Java Inheritance

- In the above example, the **eat() method is present** in both the superclass **Animal** and the subclass **Dog**.
- Here, we have created an object labrador of Dog.
- Now when we call eat() using the object labrador, the method inside Dog is called. This is because **the method inside the derived class overrides the method inside the base class**.
- This is called **method overriding**.
- Note: We have **used the @Override annotation** to tell the compiler that we are overriding a method. However, the **annotation is not mandatory.**

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.

super Keyword in Java Inheritance

- Example 3: super Keyword in Inheritance

```
class Animal {

  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }
}

// Dog inherits Animal
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");
  }
}
```

super Keyword in Java Inheritance

- Output :

```
I can eat
I eat dog food
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

- In the above example, the eat() method is present in both the base class Animal and the derived class Dog. Notice the statement,

```
super.eat();
```

- Here, the super keyword is used to call the eat() method present in the superclass.
- We can also use the super keyword to call the constructor of the superclass from the constructor of the subclass.

protected Members in Inheritance

- In Java, if a class includes protected fields and methods, then these fields and methods are accessible from the subclass of the class.

- Example 4: 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);
  }
}
```

protected Members in Inheritance

- Output :

```
I am an animal.
My name is Rocky
```

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

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

    // access protected field and method
    // using the object of subclass
    labrador.name = "Rocky";
    labrador.display();

    labrador.getInfo();
  }
}
```

protected Members in Inheritance

- In the above example, we have created a class named Animal. The class includes a protected field: name and a method: display().
- We have inherited the Dog class inherits Animal. Notice the statement,

```
labrador.name = "Rocky";
labrador.display();
```

- Here, we are able to access the protected field and method of the superclass using the labrador object of the subclass.

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

1. Single Inheritance

- In single inheritance, a single subclass extends from a single superclass. For example,

Java Single Inheritance

Types of inheritance

2. Multilevel Inheritance

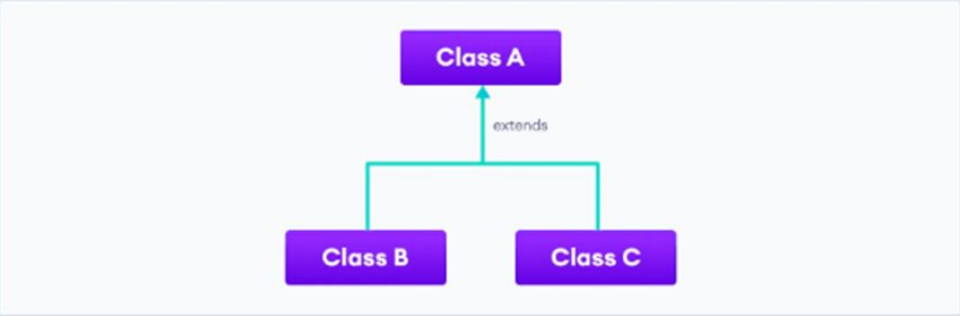
- In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,

Java Multilevel Inheritance

Types of inheritance

3. Hierarchical Inheritance

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

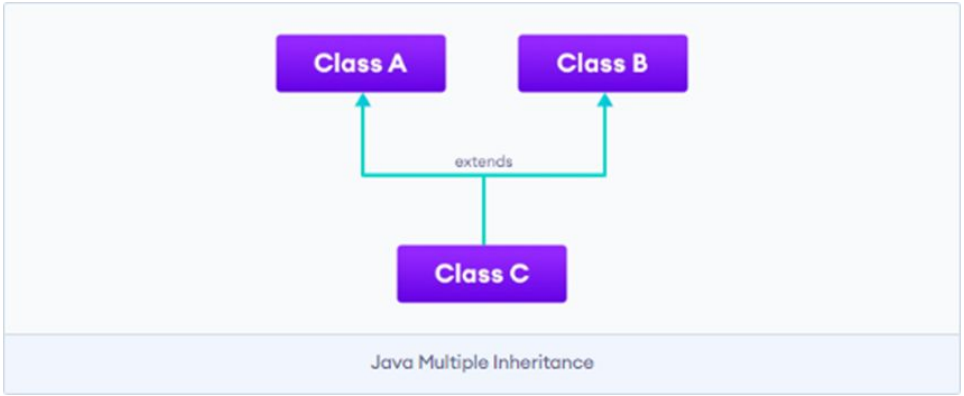


Java Hierarchical Inheritance

Types of inheritance

4. Multiple Inheritance

- In multiple inheritance, a single subclass extends from multiple superclasses. For example,
- Java doesn't support multiple inheritance. However, we can achieve multiple inheritance using interfaces.



TIA-Academy

ASSIGNMENT 01

TIA-Academy

Java Method Overriding

Java Method Overriding

- In the last tutorial, we learned about inheritance**. Inheritance is an OOP property that allows us to derive a new class (subclass) from an existing class (superclass).** The subclass inherits the attributes and methods of the superclass.
- Now, if the same method is defined in both the superclass and the subclass, then the **method of the subclass class overrides the method of the superclass**. This is known as method overriding.

Java Method Overriding

- Example 1: Method Overriding
- Output :

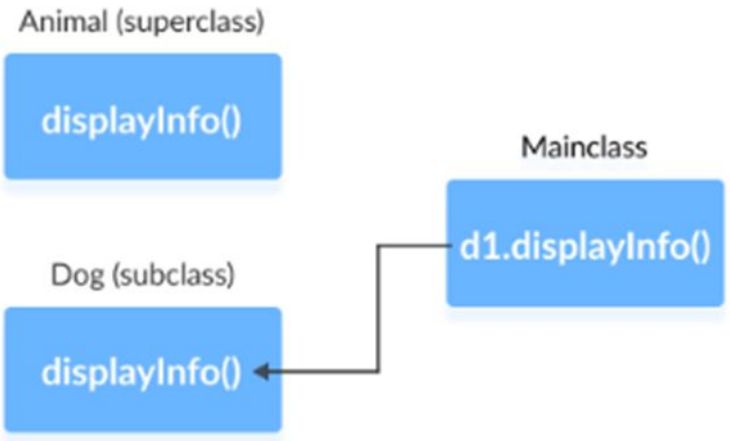
```
I am a dog.
```

- In the above program, the displayInfo() method is present in both the Animal superclass and the Dog subclass.

```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   @Override
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Java Method Overriding

- When we call displayInfo() using the d1 object (object of the subclass), the method inside the subclass Dog is called.
- The displayInfo() method of the subclass overrides the same method of the superclass.



Java Method Overriding

- Notice the use of the **@Override annotation** in our example. In Java, annotations are the metadata that we used to provide information to the compiler.
- Here, the @Override annotation specifies the compiler that the method after this annotation overrides the method of the superclass.
- It is **not mandatory to use @Override**. However, when we use this, the method should follow all the rules of overriding. Otherwise, the compiler will generate an error.

Java Overriding Rules

- Both the **superclass and the subclass must have the same method name**, the **same return type** and the **same parameter list**.
- We cannot override the method declared as final and static.
- We **should always override abstract methods** of the superclass.

super Keyword in Java Overriding

- A common question that arises while performing overriding in Java is:
- *Can we access the method of the superclass after overriding?*
- Well, the answer is **Yes**. To access the method of the superclass from the subclass, we use the super keyword.

super Keyword in Java Overriding

- Example 2: Use of super Keyword
- Output :

```
I am an animal.
I am a dog.
```

```
class Animal {
    public void displayInfo() {
        System.out.println("I am an animal.");
    }
}

class Dog extends Animal {
    public void displayInfo() {
        super.displayInfo();
        System.out.println("I am a dog.");
    }
}

class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}
```

super Keyword in Java Overriding

- In the above example, the **subclass Dog overrides** the method **displayInfo() of the superclass Animal**.
- When we call the method displayInfo() using the d1 object of the Dog subclass, the method inside the Dog subclass is called; the method inside the superclass is not called.
- Inside displayInfo() of the Dog subclass, we have used **super.displayInfo() to call displayInfo() of the superclass**.

super Keyword in Java Overriding

- It is important to note that **constructors in Java are not inherited.** Hence, there is no such thing as constructor overriding in Java.
- However**, we can call the constructor of the superclass from its subclasses**. For that, **we use super()**

Access Specifiers in Method Overriding

- The **same method declared in the superclass** and its **subclasses can have different access specifiers**. However, there is a restriction.
- We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example,
- Suppose, a method myClass() in the superclass is declared **protected**. Then, the same method myClass() in the subclass can be either **public** or **protected**, but not **private**.

Access Specifiers in Method Overriding

- Example 3: Access Specifier in Overriding
- Output :

```
I am a dog.
```

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

class Dog extends Animal {
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Access Specifiers in Method Overriding

- In the above example, the subclass Dog overrides the method displayInfo() of the superclass Animal.
- Whenever we call displayInfo() using the d1 (object of the subclass), the method inside the subclass is called.
- Notice that, the displayInfo() is declared protected in the Animal superclass. The same method has the public access specifier in the Dog subclass. This is possible because the **public provides larger access than the protected.**

TIA-Academy

ASSIGNMENT 02

TIA-Academy

Java super Keyword

Java super

- The super keyword in Java **is used in subclasses** to **access superclass members** (attributes, constructors and methods).
- Before we learn about the super keyword, make sure to know about Java inheritance.

Uses of super keyword

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

Access Overridden Methods of the

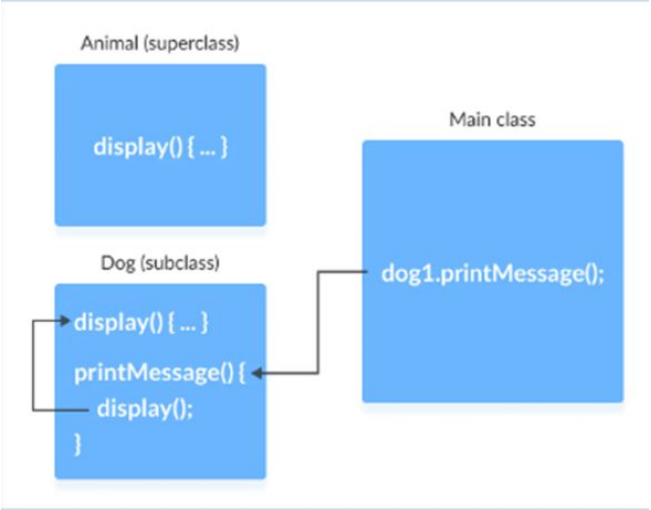
- If methods with the same name are defined in both superclass and subclass, the method in the subclass overrides the method in the superclass. This is called method overriding.
- Example 1: Method overriding
- Output:

```
I am a dog
```

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

class Dog extends Animal {
  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Important Points

- In this example, by making an object dog1 of Dog class, we can call its method printMessage() which then executes the display() statement.
- Since display() is defined in both the classes, the method of subclass Dog overrides the method of superclass Animal. Hence, the display() of the subclass is called.



Access Overridden Methods of the

- **What if the overridden method of the superclass has to be called?**
- We use super.display() if the overridden method display() of superclass Animal needs to be called.

Access Overridden Methods of the

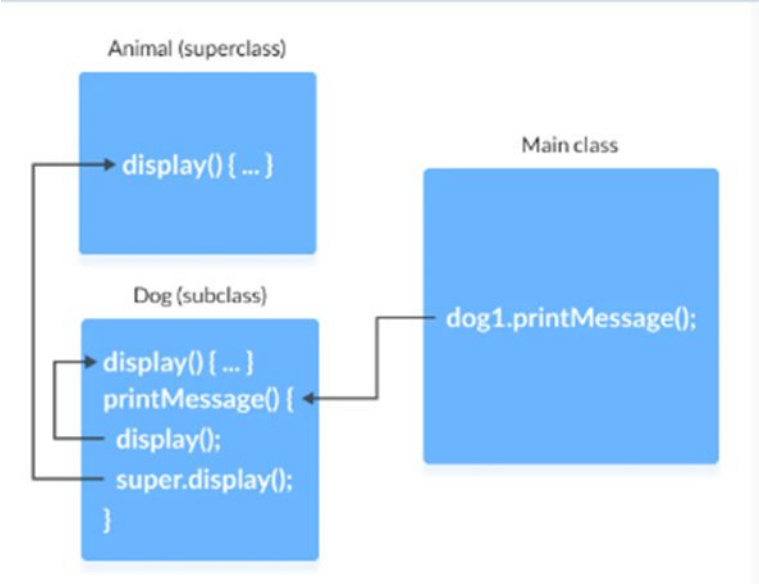
- Example 2: super to Call Superclass Method
- Output:

```
I am a dog
I am an animal
```

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

class Dog extends Animal {
  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }
  public void printMessage(){
    // this calls overriding method
    display();

    // this calls overridden method
    super.display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Access Overridden Methods of the

- Here, how the above program works.



Access Attributes of the Superclass

- The superclass and subclass can have attributes with the same name. We use the super keyword to access the attribute of the superclass.
- Example 3: Access superclass attribute
- Output:

```
I am a mammal
I am an animal
```

```
class Animal {
  protected String type="animal";
}

class Dog extends Animal {
  public String type="mammal";

  public void printType() {
    System.out.println("I am a " + type);
    System.out.println("I am an " + super.type);
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}
```

Access Attributes of the Superclass

- In this example, we have defined the same instance field **type** in both the superclass Animal and the subclass Dog.
- We then created an object dog1 of the Dog class. Then, the printType() method is called using this object.

Access Attributes of the Superclass

- Inside the printType() function,
  - **type** refers to the attribute of the **subclass Dog**.
  - **super.type** refers to the attribute of **the superclass Animal**.
- Hence, System.out.println("I am a " + type); prints I am a mammal.
- And, System.out.println("I am an " + super.type); prints I am an animal.

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.

Use of super() to access superclass constructor

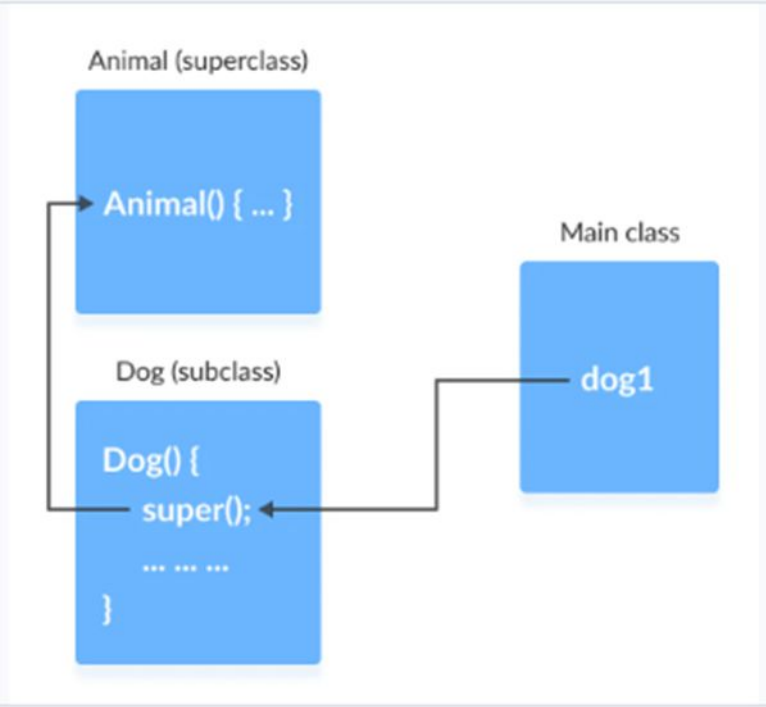
- Example 4: Use of super()
- Output:

```
I am an animal
I am a dog
```

```
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();
  }
}
```

Use of super() to access superclass constructor

- Here, when an object dog1 of Dog class is created, it automatically calls the default or no-arg constructor of that class.
- Inside the subclass constructor, the super() statement calls the constructor of the superclass and executes the statements inside it. Hence, we get the output I am an animal.



Use of super() to access superclass constructor

- The flow of the program then returns back to the subclass constructor and executes the remaining statements. Thus, I am a dog will be printed.
- However, using super() is not compulsory. Even if super() is not used in the subclass constructor, the compiler implicitly calls the default constructor of the superclass.
- So, why use redundant code if the compiler automatically invokes super()?
- It is required if the parameterized constructor (a constructor that takes arguments) of the superclass has to be called from the subclass constructor.
- The parameterized super() must always be the first statement in the body of the constructor of the subclass, otherwise, we get a compilation error.

Use of super() to access superclass constructor

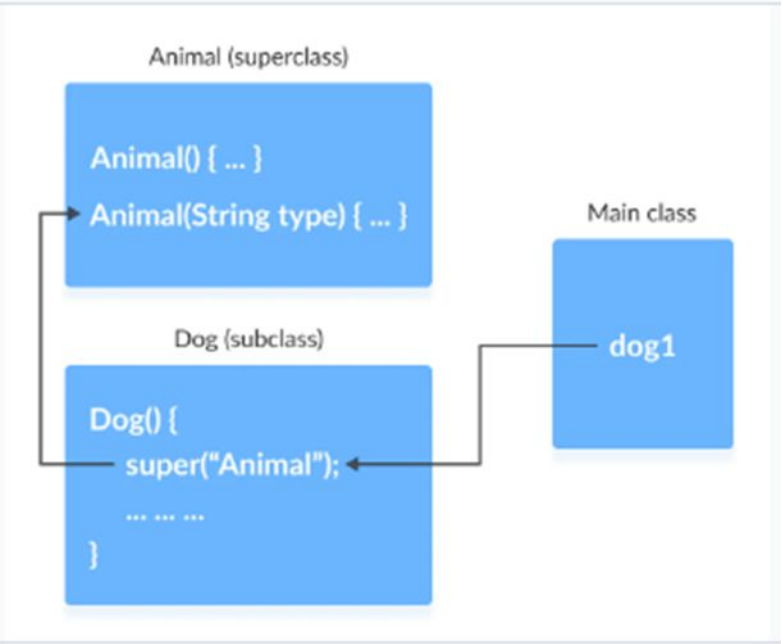
- Example 5: Call Parameterized Constructor Using super()
- Output:

```
Type: Animal
I am a dog
```

```
class Animal {
  // default or no-arg constructor
  Animal() {
    System.out.println("I am an animal");
  }
  // parameterized constructor
  Animal(String type) {
    System.out.println("Type: "+type);
  }
}

class Dog extends Animal {
  // default constructor
  Dog() {
    // calling parameterized constructor of the superclass
    super("Animal");

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Use of super() to access superclass constructor

- The compiler can automatically call the no-arg constructor. However, it cannot call parameterized constructors.
- If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.
- Note that in the example, we explicitly called the parameterized constructor super("Animal"). The compiler does not call the default constructor of the superclass in this case.



TIA-Academy

ASSIGNMENT 03
(HOME ASSIGNMENT)

TIA-Academy

Thank You