Java Nested and Inner Class

In Java, you can define a class within another class. Such class is known as `nested class`. For example,

```
class OuterClass {
    // ...
    class NestedClass {
        // ...
    }
}
```

There are two types of nested classes you can create in Java.

- Non-static nested class (inner class)
- Static nested class

Let's first look at non-static nested classes.

Non-Static Nested Class (Inner Class)

A non-static nested class is a class within another class. It has access to members of the enclosing class (outer class). It is commonly known as inner class.

Since the inner class exists within the outer class, you must instantiate the outer class first, in order to instantiate the inner class.

Here's an example of how you can declare inner classes in Java.

Example 1: Inner class

```
class CPU {
    double price;
    // nested class
    class Processor{

        // members of nested class
        double cores;
        String manufacturer;

        double getCache(){
            return 4.3;
        }
    }
```

```
    // nested protected class
    protected class RAM{

        // members of protected nested class
        double memory;
        String manufacturer;

        double getClockSpeed(){
            return 5.5;
        }
    }
}

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

        // create object of Outer class CPU
        CPU cpu = new CPU();

       // create an object of inner class Processor using outer class
        CPU.Processor processor = cpu.new Processor();

        // create an object of inner class RAM using outer class CPU
        CPU.RAM ram = cpu.new RAM();
        System.out.println("Processor Cache = " +
processor.getCache());
        System.out.println("Ram Clock speed = " +
ram.getClockSpeed());
    }
}
```

Output:

```
Processor Cache = 4.3
Ram Clock speed = 5.5
```

In the above program, there are two nested classes: *Processor* and *RAM* inside the outer class: CPU. We can declare the inner class as protected. Hence, we have declared the RAM class as protected.

Inside the Main class,

- we first created an instance of an outer class CPU named cpu.
- Using the instance of the outer class, we then created objects of inner classes:

```
CPU.Processor processor = cpu.new Processor;

  CPU.RAM ram = cpu.new RAM();
```

Note: We use the dot (.) operator to create an instance of the inner class using the outer class

Accessing Members of Outer Class within Inner Class

We can access the members of the outer class by using this keyword.

Example 2: Accessing Members

```
class Car {
    String carName;
    String carType;

    // assign values using constructor
    public Car(String name, String type) {
        this.carName = name;
        this.carType = type;
    }

    // private method
    private String getCarName() {
        return this.carName;
    }

// inner class
    class Engine {
        String engineType;
        void setEngine() {

           // Accessing the carType property of Car
            if(Car.this.carType.equals("4WD")){

                // Invoking method getCarName() of Car
                if(Car.this.getCarName().equals("Crysler")) {
                    this.engineType = "Smaller";
                } else {
                    this.engineType = "Bigger";
                }

            }else{
                this.engineType = "Bigger";
            }
        }
        String getEngineType(){
            return this.engineType;
        }
```

```
    }
}

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

// create an object of the outer class Car
        Car car1 = new Car("Mazda", "8WD");

        // create an object of inner class using the outer class
        Car.Engine engine = car1.new Engine();
        engine.setEngine();
        System.out.println("Engine Type for 8WD= " +
engine.getEngineType());

        Car car2 = new Car("Crysler", "4WD");
        Car.Engine c2engine = car2.new Engine();
        c2engine.setEngine();
        System.out.println("Engine Type for 4WD = " +
c2engine.getEngineType());
    }
}
```

Output:

```
Engine Type for 8WD= Bigger
Engine Type for 4WD = Smaller
```

In the above program, we have the inner class named *Engine* inside the outer class *Car*. Here, notice the line,

```
if(Car.this.carType.equals("4WD")) {...}
```

We are using this keyword to access the *carType* variable of the outer class. You may have noticed that instead of using this.carType we have used Car.this.carType.

It is because if we had not mentioned the name of the outer class *Car*, then this keyword will represent the member inside the inner class.

Similarly, we are also accessing the method of the outer class from the inner class.

```
if (Car.this.getCarName().equals("Crysler") {...}
```

It is important to note that, although the getCarName() is a private method, we are able to access it from the inner class.

Static Nested Class

In Java, we can also define a static class inside another class. Such class is known as static nested class. Static nested classes are not called static inner classes.

Unlike inner class, a static nested class cannot access the member variables of the outer class. It is because the **static nested class** doesn't require you to create an instance of the outer class.

```
OuterClass.NestedClass obj = new OuterClass.NestedClass();
```

Here, we are creating an object of the **static nested class** by simply using the class name of the outer class. Hence, the outer class cannot be referenced using OuterClass.this.

Example 3: Static Inner Class

```
class MotherBoard {

   // static nested class
   static class USB{
       int usb2 = 2;
       int usb3 = 1;
       int getTotalPorts(){
           return usb2 + usb3;
       }
   }

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

       // create an object of the static nested class
       // using the name of the outer class
       MotherBoard.USB usb = new MotherBoard.USB();
       System.out.println("Total Ports = " + usb.getTotalPorts());
   }
}
```

Output:

```
Total Ports = 3
```

In the above program, we have created a static class named *USB* inside the class *MotherBoard*. Notice the line,

```
MotherBoard.USB usb = new MotherBoard.USB();
```

Here, we are creating an object of *USB* using the name of the outer class.

Now, let's see what would happen if you try to access the members of the outer class:

Example 4: Accessing members of Outer class inside Static Inner Class

```
class MotherBoard {
   String model;
   public MotherBoard(String model) {
       this.model = model;
   }

   // static nested class
   static class USB{
       int usb2 = 2;
       int usb3 = 1;
       int getTotalPorts(){
           // accessing the variable model of the outer classs
           if(MotherBoard.this.model.equals("MSI")) {
               return 4;
           }
           else {
               return usb2 + usb3;
           }
       }
   }
}
public class Main {
   public static void main(String[] args) {

       // create an object of the static nested class
       MotherBoard.USB usb = new MotherBoard.USB();
       System.out.println("Total Ports = " + usb.getTotalPorts());
   }
}
```

When we try to run the program, we will get an error:

```
error: non-static variable this cannot be referenced from a static
context
```

This is because we are not using the object of the outer class to create an object of the inner class. Hence, there is no reference to the outer class `Motherboard` stored in `Motherboard.this`.

Key Points to Remember

- Java treats the inner class as a regular member of a class. They are just like methods and variables declared inside a class.
- Since inner classes are members of the outer class, you can apply any access modifiers like private, protected to your inner class which is not possible in normal classes.
- Since the nested class is a member of its enclosing outer class, you can use the dot (.) notation to access the nested class and its members.
- Using the nested class will make your code more readable and provide better encapsulation.
- Non-static nested classes (inner classes) have access to other members of the outer/enclosing class, even if they are declared private.

Java Nested Static Class

As learned in previous tutorials, we can have a class inside another class in Java. Such classes are known as nested classes. In Java, nested classes are of two types:

- Nested non-static class (Inner class)
- Nested static class.

Java Nested Static Class

We use the keyword `static` to make our nested class static.

Note: In Java, only nested classes are allowed to be static.

Like regular classes, static nested classes can include both static and non-static fields and methods. For example,

```
Class Animal {
   static class Mammal {
      // static and non-static members of Mammal
   }
   // members of Animal
}
```

Static nested classes are associated with the outer class.

To access the static nested class, we don’t need objects of the outer class.

Example: Static Nested Class

```
class Animal {

// inner class
   class Reptile {
      public void displayInfo() {
        System.out.println("I am a reptile.");
      }
   }

// static class
   static class Mammal {
      public void displayInfo() {
        System.out.println("I am a mammal.");
      }
   }
}

class Main {
   public static void main(String[] args) {
      // object creation of the outer class
      Animal animal = new Animal();

      // object creation of the non-static class
      Animal.Reptile reptile = animal.new Reptile();
      reptile.displayInfo();

      // object creation of the static nested class
      Animal.Mammal mammal = new Animal.Mammal();
      mammal.displayInfo();

   }
}
```

Output

```
I am a reptile.
I am a mammal.
```

In the above program, we have two nested class *Mammal* and *Reptile* inside a class *Animal*.

To create an object of the non-static class Reptile, we have used

```
Animal.Reptile reptile = animal.new Reptile()
```

To create an object of the static class *Mammal*, we have used

```
Animal.Mammal mammal = new Animal.Mammal()
```

Accessing Members of Outer Class

In Java, static nested classes are associated with the outer class. This is why static nested classes can only access the class members (static fields and methods) of the outer class.

Let's see what will happen if we try to access non-static fields and methods of the outer class.

Example: Accessing Non-static members

```
class Animal {
  static class Mammal {
   public void displayInfo() {
     System.out.println("I am a mammal.");
   }
 }

 class Reptile {
   public void displayInfo() {
     System.out.println("I am a reptile.");
   }
 }

 public void eat() {
   System.out.println("I eat food.");
 }
}

class Main {
 public static void main(String[] args) {
   Animal animal = new Animal();
   Animal.Reptile reptile = animal.new Reptile();
   reptile.displayInfo();

   Animal.Mammal mammal = new Animal.Mammal();
   mammal.displayInfo();
   mammal.eat();
 }
}
```

Output

```
Main.java:28: error: cannot find symbol
    mammal.eat();
           ^
  symbol:   method eat()
  location: variable mammal of type Mammal
1 error
```

```
compiler exit status 1
```

In the above example, we have created a non-static method eat() inside the class *Animal*.

Now, if we try to access eat() using the object *mammal*, the compiler shows an error.

It is because *mammal* is an object of a static class and we cannot access non-static methods from static classes.

Static Top-level Class

As mentioned above, only nested classes can be static. We cannot have static top-level classes.

Let's see what will happen if we try to make a top-level class static.

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

class Main {
 public static void main(String[] args) {
   Animal.displayInfo();
 }
}
```

Output

```
Main.java:1: error: modifier static not allowed here
static class Animal {
       ^
1 error
compiler exit status 1
```

In the above example, we have tried to create a static class *Animal*. Since Java doesn't allow static top-level class, we will get an error.

Java Anonymous Class

In Java, a class can contain another class known as nested class. It's possible to create a nested class without giving any name.

A nested class that doesn't have any name is known as an anonymous class.

An anonymous class must be defined inside another class. Hence, it is also known as an anonymous inner class. Its syntax is:

```
class outerClass {

    // defining anonymous class
    object1 = new Type(parameterList) {
         // body of the anonymous class
    };
}
```

Anonymous classes usually extend subclasses or implement interfaces.

Here, Type can be

1. a superclass that an anonymous class extends
2. an interface that an anonymous class implements

The above code creates an object, object1, of an anonymous class at runtime.

Note: Anonymous classes are defined inside an expression. So, the semicolon is used at the end of anonymous classes to indicate the end of the expression.

Example 1: Anonymous Class Extending a Class

```
class Polygon {
   public void display() {
      System.out.println("Inside the Polygon class");
   }
}

class AnonymousDemo {
   public void createClass() {

      // creation of anonymous class extending class Polygon
      Polygon p1 = new Polygon() {
         public void display() {
            System.out.println("Inside an anonymous class.");
         }
```

```
        };
        p1.display();
    }
}

class Main {
    public static void main(String[] args) {
         AnonymousDemo an = new AnonymousDemo();
         an.createClass();
    }
}
```

Output

```
Inside an anonymous class.
```

In the above example, we have created a class *Polygon*. It has a single method display().

We then created an anonymous class that extends the class *Polygon* and overrides the display() method.

When we run the program, an object *p1* of the anonymous class is created. The object then calls the display() method of the anonymous class.

Example 2: Anonymous Class Implementing an Interface

```
interface Polygon {
    public void display();
}

class AnonymousDemo {
    public void createClass() {

        // anonymous class implementing interface
        Polygon p1 = new Polygon() {
            public void display() {
                System.out.println("Inside an anonymous class.");
            }
        };
        p1.display();
    }
}

class Main {
    public static void main(String[] args) {
        AnonymousDemo an = new AnonymousDemo();
        an.createClass();
    }
```

```
}
```

Output

```
Inside an anonymous class.
```

In the above example, we have created an anonymous class that implements the Polygon interface.

Advantages of Anonymous Classes

In anonymous classes, objects are created whenever they are required. That is, objects are created to perform some specific tasks. For example,

```
Object = new Example() {
   public void display() {
      System.out.println("Anonymous class overrides the method
display().");
   }
};
```

Here, an object of the anonymous class is created dynamically when we need to override the display() method.

Anonymous classes also help us to make our code concise.

Java Singleton

Java Singleton ensures that only one object of a class can be created.

Here's how we can implement singletons in Java.

- create a private constructor that restricts to create an object outside of the class
- create a private attribute that refers to the singleton object.
- create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

Java Singleton Example

```
class SingletonExample {

// private field that refers to the object
   private static SingletonExample singleObject;

   private SingletonExample() {
      // constructor of the SingletonExample class
   }

   public static SingletonExample getInstance() {
      // write code that allows us to create only one object
      // access the object as per our need
   }
}
```

In the above example,

- private static SingletonExample singleObject - a reference to the object of the class.
- private SingletonExample() - a private constructor that restricts creating objects outside of the class.
- public static SingletonExample getInstance() - this method returns the reference to the only object of the class. Since the method static, it can be accessed using the class name.

Use of Singleton in Java

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

```
class Database {
   private static Database dbObject;

   private Database() {
   }

   public static Database getInstance() {

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

       // returns the singleton object
```

```
        return dbObject;
    }

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

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

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

        db1.getConnection();
    }
}
```

When we run the program, the output will be:

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

In our above example,

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

Singleton is a design pattern rather than a feature specific to Java. A design pattern is like our code library that includes various coding techniques shared by programmers around the world.

It's important to note that, there are only a few scenarios (like logging) where singletons make sense. We recommend you avoid using singletons completely if you are not sure whether to use them or not.

Java enums

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

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

Here, we have created an enum named *Size*. It contains fixed values *SMALL*, *MEDIUM*, *LARGE*, and *EXTRALARGE*.

These values inside the braces are called enum constants (values).

Note: The enum constants are usually represented in uppercase.

Example 1: Java Enum

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

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

Output

```
SMALL
MEDIUM
```

As we can see from the above example, we use the enum name to access the constant values.

Also, we can create variables of enum types. For example,

Size pizzaSize;

Here, *pizzaSize* is a variable of the *Size* type. It can only be assigned with 4 values.

```
pizzaSize = Size.SMALL;
pizzaSize = Size.MEDIUM;
pizzaSize = Size.LARGE;
pizzaSize = Size.EXTRALARGE;
```

Example 2: Java Enum with the switch statement

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

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

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

Output

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

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

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

Notice the statement,

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

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

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

Enum Class in Java

In Java, enum types are considered to be a special type of class. It was introduced with the release of Java 5.

An enum class can include methods and fields just like regular classes.

```
enum Size {
    constant1, constant2, …, constantN;

    // methods and fields
}
```

When we create an enum class, the compiler will create instances (objects) of each enum constants. Also, all enum constant is always public static final by default.

Example 3: Java Enum Class

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

  public String getSize() {

    // this will refer to the object SMALL
    switch(this) {
      case SMALL:
        return "small";

      case MEDIUM:
        return "medium";

      case LARGE:
        return "large";

      case EXTRALARGE:
        return "extra large";

      default:
        return null;
```

```
      }
    }

  public static void main(String[] args) {

    // call getSize()
    // using the object SMALL
    System.out.println("The size of the pizza is " +
Size.SMALL.getSize());
  }
}
```

Output

```
The size of the pizza is small
```

In the above example, we have created an enum class *Size*. It has four constants *SMALL*, *MEDIUM*, *LARGE* and *EXTRALARGE*.

Since *Size* is an enum class, the compiler automatically creates instances for each enum constants.

Here inside the main() method, we have used the instance *SMALL* to call the getSize() method.

Note: Like regular classes, an enum class also may include constructors.

Methods of Java Enum Class

There are some predefined methods in enum classes that are readily available for use.

1. Java Enum ordinal()

The ordinal() method returns the position of an enum constant. For example,

```
ordinal(SMALL)
// returns 0
```

2. Enum compareTo()

The compareTo() method compares the enum constants based on their ordinal value. For example,

```
Size.SMALL.compareTo(Size.MEDIUM)
 // returns ordinal(SMALL) - ordinal(MEDIUM)
```

3. Enum toString()

The toString() method returns the string representation of the enum constants. For example,

```
SMALL.toString()
// returns "SMALL"
```

4. Enum name()

The name() method returns the defined name of an enum constant in string form. The returned value from the name() method is final. For example,

```
name(SMALL)
// returns "SMALL"
```

5. Java Enum valueOf()

The valueOf() method takes a string and returns an enum constant having the same string name. For example,

```
Size.valueOf("SMALL")
// returns constant SMALL.
```

6. Enum values()

The values() method returns an array of enum type containing all the enum constants. For example,

```
Size[] enumArray = Size.value();
```

Why Java Enums?

In Java, enum was introduced to **replace the use of int constants**.

Suppose we have used a collection of int constants.

```
class Size {
   public final static int SMALL = 1;
   public final static int MEDIUM = 2;
```

```
   public final static int LARGE = 3;
   public final static int EXTRALARGE = 4;
}
```

Here, the problem arises if we print the constants. It is because only the number is printed which might not be helpful.

So, instead of using int constants, we can simply use enums. For example,

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

This makes our code more intuitive.

Also, enum provides **compile-time type safety**.

If we declare a variable of the *Size* type. For example,

```
Size size;
```

Here, it is guaranteed that the variable will hold one of the four values. Now, If we try to pass values other than those four values, the compiler will generate an error.

Java enum Constructor

Before you learn about enum constructors, make sure to know about Java enums.

In Java, an enum class may include a constructor like a regular class. These enum constructors are either

- **private** - accessible within the class
  or
- **package-private** - accessible within the package

Example: enum Constructor

```
enum Size {

   // enum constants calling the enum constructors
   SMALL("The size is small."),
   MEDIUM("The size is medium."),
   LARGE("The size is large."),
   EXTRALARGE("The size is extra large.");
```

```
   private final String pizzaSize;

   // private enum constructor
   private Size(String pizzaSize) {
      this.pizzaSize = pizzaSize;
   }

   public String getSize() {
      return pizzaSize;
   }
}

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

Output

```
The size is small.
```

In the above example, we have created an enum *Size*. It includes a private enum constructor. The constructor takes a string value as a parameter and assigns value to the variable *pizzaSize*.

Since the constructor is private, we cannot access it from outside the class. However, we can use enum constants to call the constructor.

In the *Main* class, we assigned SMALL to an enum variable *size*. The constant *SMALL* then calls the constructor *Size* with string as an argument.

Finally, we called getSize() using *size*.

Java enum Strings

Before you learn about enum strings, make sure to know about Java enum.

In Java, we can get the string representation of enum constants using the toString() method or the name() method. For example,

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

```
}

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

      System.out.println("string value of SMALL is " +
Size.SMALL.toString());
      System.out.println("string value of MEDIUM is " +
Size.MEDIUM.name());

   }
}
```

Output

```
string value of SMALL is SMALL
string value of MEDIUM is MEDIUM
```

In the above example, we have seen the default string representation of an enum constant is the name of the same constant.

Change Default String Value of enums

We can change the default string representation of enum constants by overriding the toString() method. For example,

```
enum Size {
   SMALL {

      // overriding toString() for SMALL
      public String toString() {
        return "The size is small.";
      }
   },

   MEDIUM {

     // overriding toString() for MEDIUM
      public String toString() {
        return "The size is medium.";
      }
   };
}

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

Output

```
The size is medium.
```

In the above program, we have created an enum *Size*. And we have overridden the toString() method for enum constants SMALL and MEDIUM.

Note: We cannot override the name() method. It is because the name() method is final.

Java Reflection

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

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

Reflection of Java Classes

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

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

There exists three ways to create objects of Class:

1. Using forName() method

```
class Dog {...}

// create object of Class
// to reflect the Dog class
Class a = Class.forName("Dog");
```

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

2. Using getClass() method

```
// create an object of Dog class
Dog d1 = new Dog();
```

```
// create an object of Class
// to reflect Dog
Class b = d1.getClass();
```

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

3. Using .class extension

```
// create an object of Class
// to reflect the Dog class
Class c = Dog.class;
```

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

Example: Java Class Reflection

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

class Animal {
}

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

// put this in Main.java file
class Main {
  public static void main(String[] args) {
    try {
      // create an object of Dog
      Dog d1 = new Dog();

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

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

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

      // convert the access modifier to string
      String mod = Modifier.toString(modifier);
```

```
      System.out.println("Modifier: " + mod);

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

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

Output

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

In the above example, we have created a superclass: Animal and a subclass: Dog. Here, we are trying to inspect the class Dog.

Notice the statement,

```
Class obj = d1.getClass();
```

Here, we are creating an object obj of Class using the getClass() method. Using the object, we are calling different methods of Class.

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

Note: We are using the Modifier class to convert the integer access modifier to a string.

Reflecting Fields, Methods, and Constructors

The package java.lang.reflect provides classes that can be used for manipulating class members. For example,

- **Method class** - provides information about methods in a class
- **Field class** - provides information about fields in a class
- **Constructor class** - provides information about constructors in a class

1. Reflection of Java Methods

The Method class provides various methods that can be used to get information about the methods present in a class. For example,

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

class Dog {

  // methods of the class
  public void display() {
    System.out.println("I am a dog.");
  }

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

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

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

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

      // using object of Class to
      // get all the declared methods of Dog
      Method[] methods = obj.getDeclaredMethods();

      // create an object of the Method class
      for (Method m : methods) {

        // get names of methods
        System.out.println("Method Name: " + m.getName());

        // get the access modifier of methods
        int modifier = m.getModifiers();
        System.out.println("Modifier: " +
Modifier.toString(modifier));

        // get the return types of method
        System.out.println("Return Types: " + m.getReturnType());
        System.out.println(" ");
      }
```

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

Output

```
Method Name: display
Modifier: public
Return Types: void

Method Name: makeSound
Modifier: private
Return Types: void
```

In the above example, we are trying to get information about the methods present in the Dog class. As mentioned earlier, we have first created an object obj of Class using the getClass() method.

Notice the expression,

```
Method[] methods = obj.getDeclaredMethod();
```

Here, the getDeclaredMethod() returns all the methods present inside the class.

Also, we have created an object m of the Method class. Here,

- m.getName() - returns the name of a method
- m.getModifiers() - returns the access modifier of methods in integer form
- m.getReturnType() - returns the return type of methods

The Method class also provides various other methods that can be used to inspect methods at run time.

2. Reflection of Java Fields

Like methods, we can also inspect and modify different fields of a class using the methods of the Field class. For example,

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

class Dog {
```

```
  public String type;
}

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

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

      // access and set the type field
      Field field1 = obj.getField("type");
      field1.set(d1, "labrador");

      // get the value of the field type
      String typeValue = (String) field1.get(d1);
      System.out.println("Value: " + typeValue);

      // get the access modifier of the field type
      int mod = field1.getModifiers();

      // convert the modifier to String form
      String modifier1 = Modifier.toString(mod);
      System.out.println("Modifier: " + modifier1);
      System.out.println(" ");
    }

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

Output

```
Value: labrador
Modifier: public
```

In the above example, we have created a class named Dog. It includes a public field named type. Notice the statement,

```
Field field1 = obj.getField("type");
```

Here, we are accessing the public field of the Dog class and assigning it to the object field1 of the Field class.

We then used various methods of the Field class:

- **field1.set()** - sets the value of the field
- **field1.get()** - returns the value of field
- **field1.getModifiers()** - returns the value of the field in integer form

Similarly, we can also access and modify private fields as well. However, the reflection of private field is little bit different than the public field. For example,

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

class Dog {
  private String color;
}

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

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

      // access the private field color
      Field field1 = obj.getDeclaredField("color");

      // allow modification of the private field
      field1.setAccessible(true);

      // set the value of color
      field1.set(d1, "brown");

      // get the value of field color
      String colorValue = (String) field1.get(d1);
      System.out.println("Value: " + colorValue);

      // get the access modifier of color
      int mod2 = field1.getModifiers();

      // convert the access modifier to string
      String modifier2 = Modifier.toString(mod2);
      System.out.println("Modifier: " + modifier2);
    }

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

```
    }
  }
}
```

Output

```
Value: brown
Modifier: private
```

In the above example, we have created a class named *Dog*. The class contains a private field named *color*. Notice the statement.

```
Field field1 = obj.getDeclaredField("color");

field1.setAccessible(true);
```

Here, we are accessing *color* and assigning it to the object *field1* of the Field class. We then used *field1* to modify the accessibility of *color* and allows us to make changes to it.

We then used field1 to perform various operations on the private field color.

3. Reflection of Java Constructor

We can also inspect different constructors of a class using various methods provided by the Constructor class. For example,

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

class Dog {

  // public constructor without parameter
  public Dog() {

  }

  // private constructor with a single parameter
  private Dog(int age) {

  }

}

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

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

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

      // get all constructors of Dog
      Constructor[] constructors = obj.getDeclaredConstructors();

      for (Constructor c : constructors) {

        // get the name of constructors
        System.out.println("Constructor Name: " + c.getName());

        // get the access modifier of constructors
        // convert it into string form
        int modifier = c.getModifiers();
        String mod = Modifier.toString(modifier);
        System.out.println("Modifier: " + mod);

        // get the number of parameters in constructors
        System.out.println("Parameters: " + c.getParameterCount());
        System.out.println("");
      }
    }

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

Output

```
Constructor Name: Dog
Modifier: public
Parameters: 0

Constructor Name: Dog
Modifier: private
Parameters: 1
```

In the above example, we have created a class named Dog. The class includes two constructors.

We are using reflection to find the information about the constructors of the class. Notice the statement,

```
Constructor[] constructors = obj.getDeclaredConstructor();
```

Here, the we are accessing all the constructors present in *Dog* and assigning them to an array *constructors* of the `Constructor` type.

We then used object *c* to get different informations about the constructor.

- **c.getName()** - returns the name of the constructor
- **c.getModifiers()** - returns the access modifiers of the constructor in integer form
- **c.getParameterCount()** - returns the number of parameters present in each constructor