Classes And Objects

22 May 2024 20:35

Agenda :

1. Classes
2. Objects
3. More on Classes
4. Nested Classes
5. Lambda Expressions
6. Enum Types

1. Classes

- Declaring A Class

```
class MyClass {
   // field, constructor, and
   // method declarations
}
```

This is a *class declaration*.

- Declaring Member Variables (use Camel Case to define Variables)

Variables are of multiple kinds :

- Member Variables : called **Fields.**
- Variables in a method : **local Variable.**
- Variable in method Declaration : called **Parameters.**

- Defining Methods

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
   //do the calculation here
}
```

- Method Overloading

Method Overloading is a technique in which methods within a class **can have the same name if they have different parameter lists. Return type is always same.**

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

```
public class DataArtist {
   ...
   public void draw(String s) {
      ...
   }
   public void draw(int i) {
      ...
   }
   public void draw(double f) {
      ...
   }
   public void draw(int i, double f) {
      ...
   }
}
```

- Providing Constructors

A class contains constructors that are invoked to create objects from the class blueprint.

Constructor declarations look like method declarations—except that they use the name of the class and have no return type.

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
   gear = startGear;
   cadence = startCadence;
   speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8)
```

Note: If another class cannot call a MyClass constructor, it cannot directly create MyClass objects.

- Passing Reference Data Type Argument

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
   // code to move origin of circle to x+deltaX, y+deltaY
   circle.setX(circle.getX() + deltaX);
   circle.setY(circle.getY() + deltaY);

   // code to assign a new reference to circle
   circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56)
```

2. Objects

Classes are the blueprint of objects. Objects are created of respective class.

- Creating a Object

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

The first line creates an object of the Point class, and the second and third lines each create an object of the Rectangle class.

Each of these statements has three parts (discussed in detail below):

1. **Declaration**: The code set in **bold** are all variable declarations that associate a variable name with an object type.
2. **Instantiation**: The new keyword is a Java operator that creates the object.
3. **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

- The Garbage Collector

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.

3. More On Classes

- Using this keyword

Within an instance method or a constructor, this is a reference to the *current object.*

```
//constructor
  public Point(int x, int y) {
     this.x = x;
     this.y = y;
  }
```

```
public class Rectangle {
   private int x, y;
   private int width, height;

   public Rectangle() {
      this(0, 0, 1, 1);
   }
   public Rectangle(int width, int height) {
      this(0, 0, width, height);
   }
   public Rectangle(int x, int y, int width, int height) {
      this.x = x;
      this.y = y;
      this.width = width;
      this.height = height;
   }
   ...
}
```

- Controlling Access to a Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- Introduction To static Keyword

In Java, **static keyword** is mainly used for memory management. It can be used with variables methods, blocks and nested classes. It is a keyword which is used to share the same variable or method of a given class. Basically, static is used for a constant variable or a method that is same for every instance of a class.

Static Block

In order to initialize static variable, we declare static block that executes only once.

```
// static block
static {
System.out.println("Static block initialized.");
n = j * 8;
}
```

Static Variable

When you declare a variable as static, then a single copy of the variable is created and divided among all objects at the class level.

```
static int myVariable=12;
```

Static Methods

When a method is declared with the *static* keyword, it is known as a static method. The most common example of a static method is the *main( )* method.

Methods declared as static can have the following restrictions:

- They can directly call other static methods only.
- They can access static data directly.

Static Class

A class can be made static only when it is a nested class, **Nested static class doesn't needs a reference of outer class. In this case static class cannot access non static members of outer class.**

```
public class NestedExample{
        private static String str= "Vishal"
        //Static class
        static class MyNestedClass{
                //non-static method
                public void disp(){
                        System.out.println(str);
                }
        }

        public static void main(String args[]){
                NestedExample.MyNestedClass obj = new NestedExample.MyNestedClass();
                obj.disp();
        }
}
o/p = Vishal
```

4. Nested Classes

- The Java programming language allows you to define a class within another class. Such a class is called a *nested class.*

Note :
Terminology: Nested classes are divided into two categories: non-static and static.

- Non-static nested classes are called *inner classes*.
- Nested classes that are declared static are called *static nested classes*

Why use Nested Class ?

- It is a way of logically grouping classes that are only used in one place.
- **It increases encapsulation**
- It can lead to more readable and maintainable code.

Inner Class

Inner Class is the class that us declared inside the class or interface which were mainly introduced to sum up same logically relatable classes.

There are basically three types of inner classes in java.

1. Nested Inner Class	`class Outer {`
	
	`  // Class 2`
	`  // Simple nested inner class`
	`  class Inner {`
	
	`  // show() method of inner class`
	`  public void show()`
	`  {`
	
	`  // Print statement`
	`  System.out.println("In a nested class method");`
	`  }`
	`  }`
	`}`

2. Method Local Inner Classes	`class Outer {`
	
	`  // Method inside outer class`
	`  void outerMethod()`
	` {

	// Print statement
	System.out.println("inside outerMethod");
	
	// Class 2
	// Inner class
	// It is local to outerMethod()
	class Inner {
	
	// Method defined inside inner class
	void innerMethod()
	{
	
	// Print statement whenever inner class is
	// called
	System.out.println("inside innerMethod");
	}
	}
	
	// Creating object of inner class
	Inner y = new Inner();
	
	// Calling over method defined inside it
	y.innerMethod();
	}
	}

3. Anonymous Inner Classes	class GFG {
	
	// Class implementing interface
	static Hello h = new Hello() {
	
	// Method 1
	// show() method inside main class
	public void show()
	{
	// Print statement
	System.out.println("i am in anonymous class");
	}
	};
	
	// Method 2
	// Main driver method
	public static void main(String[] args)
	{
	// Calling show() method inside main() method
	h.show();
	}
	}

5. Lambda Expressions

With zero Parameter :

```
() -> System.out.println("Zero parameter lambda");
```

With Single Parameter :

```
(p) -> System.out.println("One parameter: " + p);
```

With Multiple parameter :

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

```
When to use different classes  :
```

- **Local class:** Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).
- **Anonymous class:** Use it if you need to declare fields or additional methods.
- **Lambda expression:**
  - Use it if you are encapsulating a single unit of behaviour that you want to pass to other code. For example, you would use a lambda expression if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
  - Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).
- **Nested class:** Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.
  - Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.

6. ENUM Types

An *enum type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.

```
public enum Day {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
}

public class EnumTest {
  Day day;

  public EnumTest(Day day) {
    this.day = day;
  }

  public void tellItLikeItIs() {
    switch (day) {
      case MONDAY:
        System.out.println("Mondays are bad.");
        break;

      case FRIDAY:
        System.out.println("Fridays are better.");
        break;

      case SATURDAY: case SUNDAY:
        System.out.println("Weekends are best.");
        break;

      default:
        System.out.println("Midweek days are so-so.");
        break;
    }
  }
```