UNIT-2

Classes and Objects: Introduction, Class Declaration and Modifiers, Class Members, Declaration of Class Objects, Assigning One Object to Another, Access Control for Class Members, Accessing Private Members of Class, Constructor Methods for Class, Overloaded Constructor Methods, Nested Classes, Final Class and Methods, Passing Arguments by Value and by Reference, Keyword this**.**

Methods: Introduction, Defining Methods, Overloaded Methods, Overloaded Constructor Methods, Class Objects as Parameters in Methods, Access Control, Recursive Methods, Nesting of Methods, Overriding Methods, Attributes Final and Static.

CLASSES AND OBJECTS:

Class Declaration And Modifiers

Defining a Class

- A class is a user-defined data type with a template that serves to define its properties.
- Once the class type has been defined, we can create "variables" of that type using declarations that are similar to the basic type declarations.
- In Java, these variables are termed as instances of classes, which are the actual objects.
- Class Defines Data and Methods that manipulate the Data.

The basic form of a class definition is

```
class ClassName [extends SuperClassName]
        {
                [fields declaration]
                [methods declaration]
        }
```

Modifiers

Modifiers are keywords that you can use to change the behavior or visibility of classes, methods, and variables. They can be divided into two categories: **Access Modifiers** and **Non-Access Modifiers**.

Access Modifiers

Access modifiers determine the visibility of the class to other classes. Java provides four access levels:

1. **public**: The class is accessible from any other class.

2. **protected**: The class is accessible within its package and by subclasses.
3. **default (no modifier)**: The class is accessible only within its own package.
4. **private**: The class is accessible only within the class it is defined. Note that private is not applicable to top-level classes.

Non-Access Modifiers

Non-access modifiers provide functionality other than visibility control:

1. **final**: The class cannot be subclassed.
2. **abstract**: The class cannot be instantiated and may contain abstract methods.
3. **static**: The modifier indicates that the nested class is a static member of the outer class

Class Members

Class members include fields (variables), methods, constructors, and nested classes/interfaces.

Fields Declaration

- Data is encapsulated in a class by placing data fields inside the body of the class definition.
- These variables are called instance variables because they are created whenever an object
- of the class is instantiated.
- We can declare the instance variables exactly the same way as we declare local variables

Class Rectangle

```
{

        int length;

        int width;

}
```

- The class Rectangle contains two integer type instance variables.
- It is allowed them in one line as
- **int length,width;**

Methods Declaration

The General form of a method declaration is

type methodName(parameter-list)

{

Method-body;

}

Method declarations have four basic parts

- The name of the method(method name)
- The type of the value the method returns(type)
- A list of parameters(parameter-list)
- The body of the method

Constructors

- Java supports a special type of method called a constructor, that enables an object to initialize itself when created.
- Constructors are used to initialize instance variables.

Constructor	Method
Constructor's are used to initialize instance variables	Methods are used to do general purpose calculation
Constructor Name and Class name should be same	Constructor name and Class name may or may not same
Constructor should have neither return type or void	Method should have either return type or void
Constructors are invoked at the time of object creation	Methods are invoked after object is created.

Nested Classes/Interfaces

- Classes and interfaces defined within another class.

```
public class OuterClass
{
   public class InnerClass
{
      public void display()
 {
```

```
            System.out.println("Inner Class");
        }
    }
}
```

Example:

```
class OuterClass
{
    static int x = 10;
    int y = 20;
    private static int z = 30;
    static class Innerclass
    {
        void display()
        {
            System.out.println("x = " +x);
            System.out.println("z = "+z);
            OuterClass obj = new OuterClass();
            System.out.println("y = " + obj.y);


        }
    }
}
public class Demo {
    public static void main(String args[])
    {
        // accessing a static nested class
        OuterClass.Innerclass obj1= new OuterClass.Innerclass();

        obj1.display();
    }
```

Declaration of Class Objects

Creating an instance of a class is called declaring a class object.

Person person = new Person("John", 30);

Assigning One Object to Another

Assigning one object to another makes both references point to the same object in memory.

```
public class Person
{
   public String name;
   public int age;

   // Constructor
   public Person(String name, int age)
{
      this.name = name;
      this.age = age;
  }

   // Method to display person's details

  public void display()
{
      System.out.println("Name: " + name + ", Age: " + age);
   }

   public static void main(String[] args)
{
      // Create a Person object

      Person person1 = new Person("Alice", 25);

      System.out.println("Details of person1:");

      person1.display();

      // Assign person1 to person2

      Person person2 = person1;
```

```
        System.out.println("Details of person2 (after assignment):");

        person2.display();

        // Modify person2's details

        person2.name = "Bob";

        person2.age = 30;

        // Display details of both person1 and person2

        System.out.println("Details of person1 (after modifying person2):");
        person1.display();
        System.out.println("Details of person2 (after modifying person2):");
        person2.display();
    }
}
```

Access Control for Class Members

Access control determines the visibility of class members.
Java provides four access levels:

1. public: Accessible from any other class.
2. protected: Accessible within the same package and subclasses.
3. default (no modifier): Accessible only within the same package.
4. private: Accessible only within the same class.

1. Public Access Modifier

The public modifier allows class members to be accessed from any other class.

```
import java.util.*;
 public class Demo5
{
        public int a = 10;

        public void display()
        {
                System.out.println("Public method");

        }
}

class Maindemo
{
         public static void main(String[] args)
        {
                Demo obj = new Demo();
                System.out.println(obj.a); // Accessible
                obj.display(); // Accessible
         }
}
```

2. Protected Access Modifier

The protected modifier allows class members to be accessed within the same package and subclasses.

```
import java.util.*;
class Demo6
{
        protected int a = 20;

        protected void display()
        {
                System.out.println("Protected method");
        }
}

class Demo7 extends Demo6
{
        void display1()
        {
                System.out.println(a); // Accessible
                display(); // Accessible
        }
}

class Maindemo1
{
        public static void main(String[] args)
        {
                Demo7 obj = new Demo7();
                obj.display1(); // Access protected members via subclass
         }
}
```

Default Access Modifier

The default access modifier (no modifier) allows class members to be accessed only within the same package.

```
import java.util.*;
class DemoDefault
{
   int a = 30; // Accessible only within the same package
   void display()
  {
            System.out.println("Default method");
            }
}

class TestDefault {
   public static void main(String args[])
  {
                  DemoDefault obj = new DemoDefault();
            System.out.println("Default Field: " + obj.a);
            obj.display(); // Accessible
            }
}
```

4.Private Access Modifier

The private modifier allows class members to be accessed only within the same class.

```
  import java.util.*;

  public class Demoprivate
  {
    private int a = 40; // Accessible only within the same class
   private void display()
            {
                     System.out.println("Private method");
             }

            public void display1()
            {
             System.out.println("Private Field: " + a);
             display(); // Accessible within the same class
            }
  }
```

```
class Testprivate
{
        public static void main(String[] args)
        {
                Demoprivate obj = new Demoprivate();
                obj.display1(); // Accesses private members through public method
        }
}
```

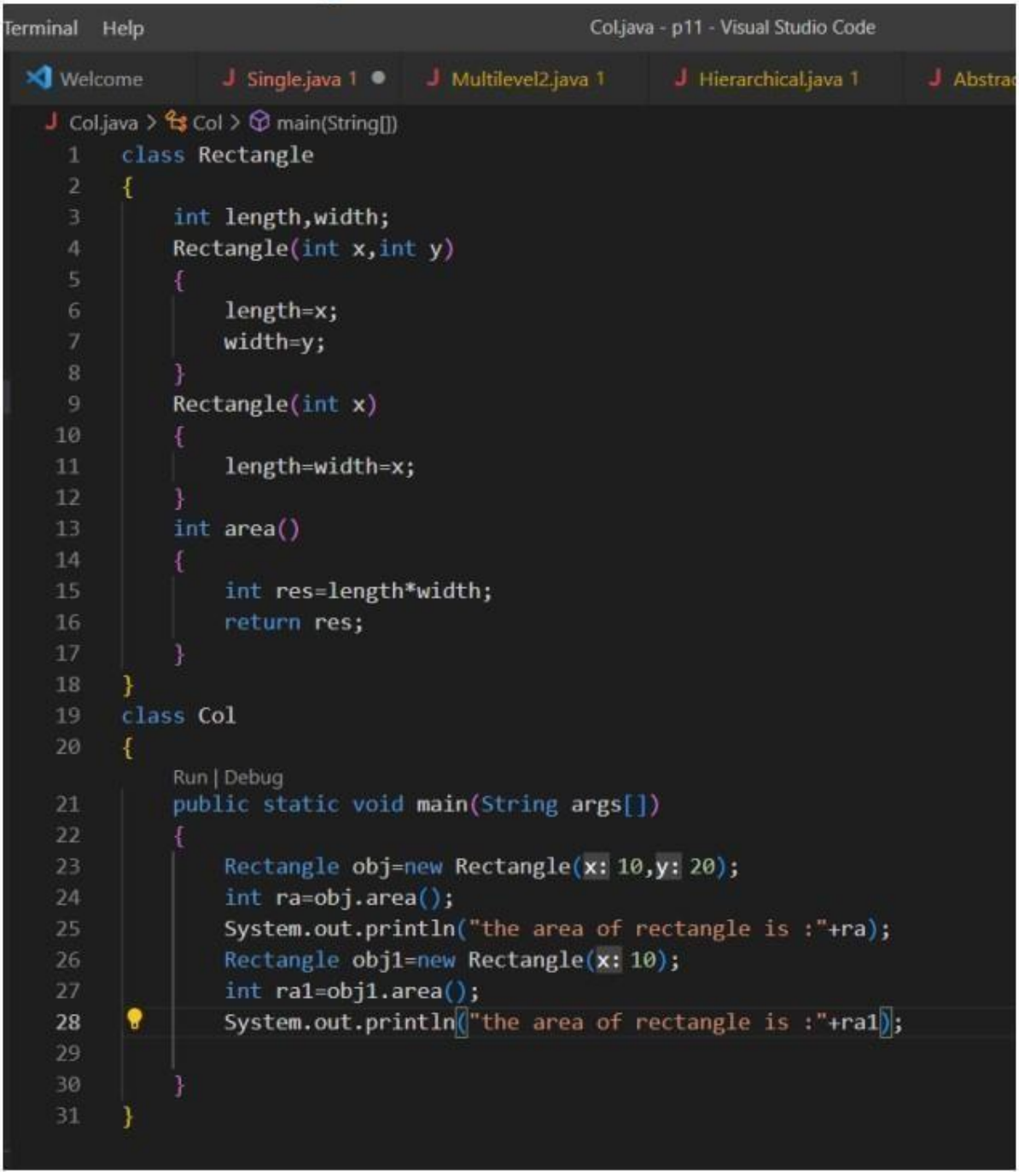
This Keyword

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

```
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);
  }
}
```

Constructor Overloading

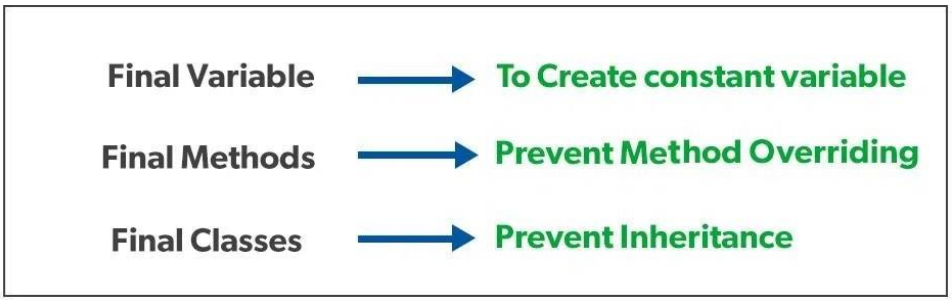
The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task



Final Class and method

The **final method** in Java is used as a **non-access modifier** applicable only to a **variable**, a **method**, or a **class.** It is used to **restrict a user** in Java**.**
The following are **different contexts** where the final is used:

1. Variable
2. Method
3. Class



Parameter Passing In Java

- There are different ways in which parameter data can be passed into and out of methods and functions.
- Let us assume that a function B() is called from another function A().
- In this case A is called the “caller function” and B is called the “called function or callee function”. Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Types of parameters

Formal Parameter: A variable and its type as they appear in the prototype of the function or method.

Syntax:
function _ name(datatype var _ name);

Actual Parameter
The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Syntax:

fun _ name(var _ name(s));

Call By Value:

- Changes made to formal parameter do not get transmitted back to the caller.
- Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment.
- This method is also called as call by value

Call by reference:

- Changes made to formal parameter do get transmitted back to the caller through parameter passing.
- Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data.
- This method is also called as **call by reference**. This method is efficient in both time and space.

Introduction to Methods

Methods in Java are blocks of code that perform specific tasks and are typically defined within classes. They encapsulate behavior and promote code reusability and modularity.

```
public class MethodsExample
{

   // Method to print a greeting message
   public void greet()
         {
             System.out.println("Hello, welcome to Java methods!");
          }

   // Method with parameters to calculate the sum of two numbers
   public int sum(int a, int b)
         {
          return a + b;
         }

   public static void main(String[] args)
```

```
        {
        MethodsExample example = new MethodsExample();

    // Calling the greet method
    example.greet();

    // Calling the sum method
    int result = example.sum(5, 3);
    System.out.println("Sum: " + result);
  }
}
```

Overloaded Methods

Overloaded methods are methods in the same class with the same name but different parameter lists (number or types), allowing flexibility in method invocation.

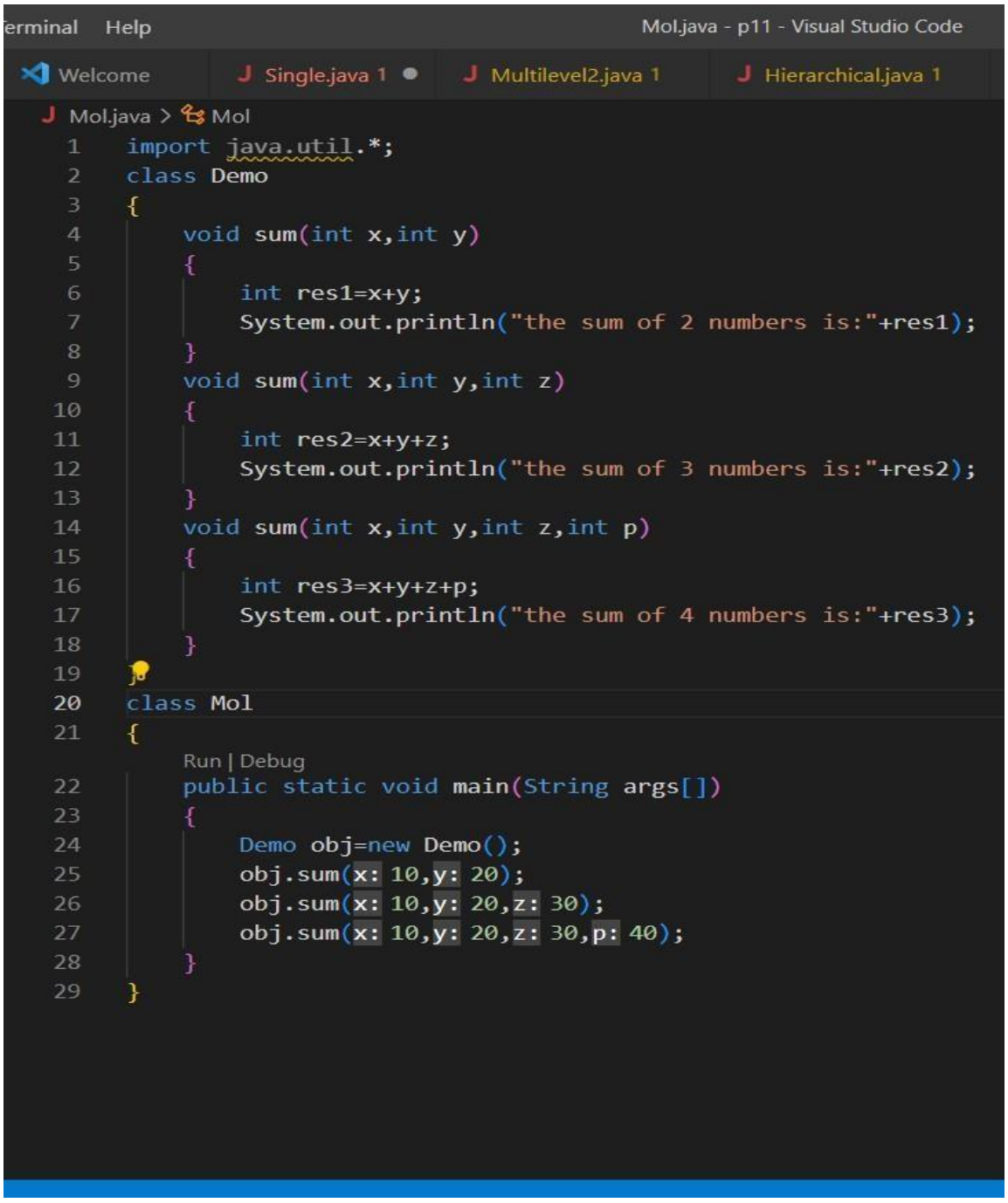
```
public class OverloadedMethodsExample
{

  // Method to add two integers
  public int add(int a, int b)
        {
              return a + b;
         }

  // Overloaded method to add three integers
  public int add(int a, int b, int c)
         {
              return a + b + c;
         }

  public static void main(String[] args)
        {
               OverloadedMethodsExample example = new OverloadedMethodsExample();

              System.out.println("Sum of two numbers: " + example.add(5, 3));
              System.out.println("Sum of three numbers: " + example.add(5, 3, 2));
        }
}
```




```
[Running] cd "f:\p11\" && javac Mol.java && java Mol
the sum of 2 numbers is:30
the sum of 3 numbers is:60
the sum of 4 numbers is:100

[Done] exited with code=0 in 1.602 seconds
```

Method overriding

Terminal Help — Mor.java - p11 - Visual Studio Code

Welcome | Single.java 1 | Multilevel2.java 1 | Hierarchical.java 1

Mor.java > ...

```java
import java.util.*;
class Rectangle
{
    double area(double l,double b)
    {
        double res=l*b;
        return res;
    }
}
class Triangle extends Rectangle
{
    double area(double l,double b)
    {
        double res1=0.5*l*b;
        return res1;
    }
}
class Mor
{
    Run | Debug
    public static void main(String args[])
    {
        Triangle obj1=new Triangle();
        double ta=obj1.area(l: 4,b: 5);
        System.out.println("the area of triangle is"+ta);
    }
}
```

```
[Running] cd "f:\p11\" && javac Mor.java && java Mor
the area of triangle is10.0

[Done] exited with code=0 in 3.822 seconds
```

Recursive Methods

Recursive methods call themselves directly or indirectly, useful for solving problems where a method repeats its behavior.

```
public class RecursiveMethodExample
{

   // Recursive method to calculate factorial
   public int factorial(int n)
         {
                  if (n == 0 || n == 1)
                  {
                           return 1;
                  }
                   else
                  {
                           return n * factorial(n - 1);
                    }
         }

   public static void main(String[] args)
         {
                  RecursiveMethodExample example = new RecursiveMethodExample();

                  // Calculate factorial of 5
                    int result = example.factorial(5);
                    System.out.println("Factorial of 5: " + result);
            }
}
```