UNIT II

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

In Java, classes and objects are basic concepts of Object Oriented Programming (OOPs) that are used to represent real-world concepts and entities.

The class represents a group of objects having similar properties and behavior. For example, the animal type Dog is a class while a particular dog named Tommy is an object of the Dog class.

Class:

- A class in Java is a set of objects which have common behavior and attributes.
- It is a user-defined blueprint from which objects are created.
- For example, Student is a class while a particular student named Ravi is an object.

Properties of Classes

- Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- Class does not occupy memory and class is called Logical Instance of object.
- Class contains variables (data) of different data types and methods.

Class Declaration in Java

```
access_modifier class <class_name>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

Access modifier may be public, private, protected, final, abstract, strictfp and even no modifier i.e. default.

Example:

```
class Rectangle
{
        int length,breadth;
        int area( )
        {
        int a = length * breadth;
        return a;
        }
}
```

Object:

- An object in Java is a basic unit of Object-Oriented Programming and represents real-world entities.
- Objects are the instances of a class that are created to use the attributes and methods of a class.
- An object consists of :
  - **State :** It is represented by attributes/properties of an object
  - **Behavior :** It is represented by the methods of an object
  - **Identity :** It gives a unique name to an object
- A typical Java program creates many objects, interact by invoking methods.
- An object occupies memory and object is called Physical Instance of the Class.

Creating Objects

Creating an object is referred to as instantiating an object from a class. In Java, objects are created using **new** operator (memory allocation operator). The **new** operator creates an object of the specified class and returns a reference to that object.

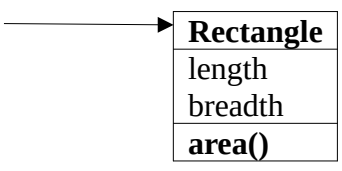
Example:

```
Rectangle r;          // object declaration
r = new Rectangle ( ); // object instantiation
```

- The first statement declares a variable to hold the object reference and it refers the **null.**

  r ⟶ **null**

- And the second statement assigns the object reference to the variable, where the object reference is the memory allocated to the object's data and methods.



The variable **r** is an object of the class **Rectangle**.

- Both the statements can also be done in single statement as follows:

  Rectangle r = new Rectangle ( );

  The method **Rectangle( )** is a default constructor of the class, this assigns the reference to the object.
- We can create any number of objects to a class.

  Rectangle r1 = new Rectangle( ) ;

  Rectangle r2 = new Rectangle( ) ;

  When multiple objects are created of same class then each object has its own separate copy of instance variables of its class which does not conflict with the instance variables of another object of same class.

Defining Methods in a Class:

- A method is a block of code that performs a specific task. It's a reusable component that can be called from other parts of your program
- A method definition comprises two components:
  - **Header:** It includes modifier, return type, name of the method and a list of parameters
  - **Body:** It is placed in braces { } and consists of declarations, executable statements and a return statement at the end.

General form of method definition:

```
modifier return_type method_name(type parameter_name,.....)
{
        ---------
        ---------
        ---------
}
```

Modifiers of the methods are as follows:

1. Access Modifiers:

a) public **b)** private **c)** protected **d)** No access modifier

- **private:** The method is accessible only within the same class.
- **default** (no modifier, also called package-private): The method is accessible within the same package but not from outside the package.
- **protected:** The method is accessible within the same package and by subclasses (even in different packages).
- **public:** The method is accessible from any other class.

2. Non - Access Modifiers:

a) static **b)** final **c)** native **d)** transient **e)** synchronized **f)** volatile

- **static:** with this modifier, a method may be called without an object of the class
- **final:** method declared with final cannot be overridden in sub class
- **native:** It indicates that the method implemented in a platform independent language like C
- **synchronized:** It is applied to make thread safe. It ensures that this method is accessed by only one thread at a time

Return Type:

A method's return type specifies the data type of the value that the method will return to the calling code. It determines what type of value the caller can expect to receive from the method.

Common Return Types:

- **void:** Indicates that the method doesn't return any value.
- **Primitive data types:** int, double, boolean, char, etc.
- **Object references:** References to objects of classes.

The return statement is used to return a value or refernce from the method back to calling code.

Parameter List:

The parameter list is enclosed in a pair of paranthesis. Each parameter is declared with its type and separated by comma from others.

Example:

```
myMethod(int n, float k, char y)
{
        --------------
        --------------
}
```

Example Method:

```
int compute(int n, int m)
{
        int k = n * m;
        return k;
}
```

ASSIGNING ONE OBJECT TO ANOTHER

When one object of a class is assigned to another object of the same class, the first object can access the second object's data and methods.

- **Let's work through an example using a** Rectangle class **to demonstrate assigning one object to another.**

```
public class Rectangle
{
   int width;
   int height;
   Rectangle(int width, int height)
   {
      this.width = width;
      this.height = height;
   }
   void area()
   {
      System.out.println("Rectangle Area: "+ (width * height));
   }
}
```

- **Now, let's create two Rectangle objects r1, r2 and assign one to the other:**

```
public class RectangleDemo
{
   public static void main(String[] args)
   {
        Rectangle r1 = new Rectangle(10, 20);
        r1.area();
        Rectangle r2 = r1; // Assigning r1 to r2
        r2.width = 30;
        r2.height = 40;
        r2.area();
       }
}
```

Output:

```
Rectangle Area: 200
Rectangle Area: 1200
```

Explanation

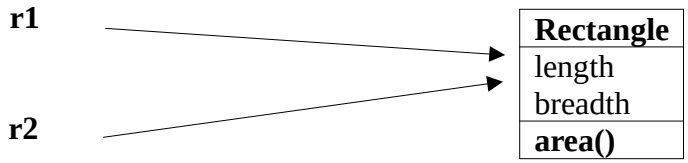
Rectangle r1 = new Rectangle(10, 20);

This statement creates the first rectangle object with width 10 and height 20.

Rectangle r2 = r1;

This statement assigns the r1 to r2. Now, both r1 and r2 point to the same object in memory.

This result in,



Effect of Modification:

When you modify r2.width and r2.height, these changes are reflected in the object that both r1 and r2 reference, because both are point to same object in memory

CLASS MODIFIERS

In Java, class modifiers are keywords that you can use to control the accessibility of a class. These access modifers are grouped as:

1. Access Modifiers
   a) No Modifier
   b) public
   c) private
   d) protected
2. Non - Access Modifiers
   a) final
   b) abstract
   c) strictfp

1. Access Modifiers

Access modifiers control the visibility of the class to other classes.

- **public :** The class is accessible from any other class. If a class is declared public , it must be in a file with the same name as the class.

```
public class MyClass
{
    // Class code
}
```

- **(default) :** When no access modifier is specified, the class has package-private access. This means the class is only accessible by other classes in the same package.

```
class MyClass
{
    // Class code
}
```

- **private and protected:** The private and protected is used only for nested class.

```
class MyClass
{
        private class x
        {
                // class code
        }
 }
```

2. Non-Access Modifiers

These modifiers control other properties of the class, such as whether it can be subclassed or if it contains abstract methods.

- **final:** A final class cannot be subclassed. This is useful when you want to prevent inheritance to protect the implementation.

```
final class MyClass
{
    // Class code
}
```

- **abstract:** An abstract class cannot be instantiated directly. It can contain abstract methods (methods without a body) that must be implemented by subclasses.

```
abstract class MyClass
{
    // Abstract method
    public abstract void doSomething();
}
```

- **strictfp:** The strictfp modifier is used to restrict floating-point calculations to ensure portability.

```
strictfp class MyClass {
    // Class code
}
```

These class modifiers are crucial in designing the architecture and accessibility of your classes in a Java application.

ACCESS CONTROL FOR CLASS MEMBERS

- In Java, access control (or access modifiers) determines the visibility and accessibility of classes, methods, variables, and constructors.
- This is crucial for ensuring that the class members are protected from unintended interference.

Access Control Levels for Class Members

Java provides four levels of access control for class members (fields, methods, constructors):

1. **private:** The member is accessible only within the same class.
2. **default** (no modifier, also called package-private): The member is accessible within the same package but not from outside the package.
3. **protected:** The member is accessible within the same package and by subclasses (even in different packages).
4. **public:** The member is accessible from any other class.

Data members are declared with access-specifiers as follows:

access_specifier datatype identifier;

examples:

private int data; // private – access to only inside the class
int data; // package-private by default
protected int data;
public int data;

ACCESSING PRIVATE DATA MEMBERS OF CLASS

In Java, class members (data fields and methods) can be declared as private to restrict their accesses/visibility to the class itself.

If one member is private, it's for a good reason: **Data Encapsulation**. This is basic object-oriented philosophy: that member is not meant to be accessed, which means this information is only relevant to this class and should not be relevant to any other class.

However, there is a way to access private members from outside the class too.

Using Getter and Setter Methods

The most conventional and recommended way to access private data fields is by using public getter and setter methods within the class.

- Setter methods are used to set/modify the values of private data members
- Getter methods are used to retrieve the value of the private data members

Program Example:

```
public class Person {
   private String name;
   public String getName()
   {
      return name;
   }
   public void setName(String name)
   {
      this.name = name;
   }
}
public class TestClass {
   public static void main(String[] args) {
      Person person = new Person();
      person.setName("John");
      System.out.println(person.getName()); // Output: John
   }
}
```

ACCESSING PRIVATE METHODS OF A CLASS

- A private method in Java is defined using the private access modifier. This means it can only be accessed within the same class.
- A public method is defined using the public access modifier. This method can be accessed from outside the class.
- In Java, to access a private methods, we have to declare a public method in the same class. Inside this public method, you can call the private method to perform some operation and return the result.

Program Illustration of accessing private methods of a class:

```
class Farm
{
private double length;
private double width;
Farm(double l, double w)
{
        length = l;
        width = w;
}
```

```
    private double area() // private method
    {
        return length * width;
    }
    public void getArea()
    {
        int a = area(); // invoking a private method from public method
        System.out.println("Area of the Farm:"+a);
    }
    public static void main(String[] args)
    {
        Farm f1 = new Farm(24,15);
        f1.getArea();
    }
}
```

In the above example class, to access the private method area(), a public method getArea() is defined, in which private method area() is invoked.

And this is a way of controlling the access and encapsulating the inner workings of the class

---

this KEYWORD

In Java, 'this' keyword is a reference variable that refers to the current object, or "this" in Java is a keyword that refers to the current object instance.

It can be used to call current class methods and fields, to pass an instance of the current class as a parameter, and to differentiate between the local and instance variables.

When a method or constructor has a parameter with the same name as an instance variable, the this keyword is used to differentiate between the instance variable and the parameter.

Program Example:

```
class Test {
    int a;
    int b;
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
    void display()
```

```
    {
        System.out.println("a = " + a + " b = " + b);
    }
    public static void main(String[] args)
    {
        Test object = new Test(10, 20);
        object.display();
    } // main ends
} // class ends
```

CLASS MEMBERS or STATIC MEMBERS

The class members are those declared in the body of the class. These may comprise data fields (variables), methods, nested classes and interfaces. The scope of the all the members extends to the entire class.

Data Fields or Variables: Data Fields comprise of two types:

1. **Non – Static Variables:** These include local variables and instance variables and varies in scope and time.
   a) ***Instance Variables:*** These varaibles are individual to an object and an object keeps a own copy of these variables in its memory. And their scope is object. One object's data variables can't be accessed by other object.
   b) ***Local Variables :*** These are local in scope and not accessible outside their scope. These are declared inside the method and they are accessible within that method only.
2. **Static Variables or Static Data Members:**
   - static variables are declared using the keyword **static**
   - The values of these variables are common to all the objects of the class. The class keeps only one copy of these variables and all the objects use the same copy. As class variables belong to the whole class, these are also called Class Variables.
   - Java created only one copy of static variable common to all instances of the class that can be allowed to use even the class is not instantiated. Therefore, the static variables are accessed without using the objects.
   - Static variables are initialized when the class is first loaded into JVM memory

Program Example: illustration of difference between instance and static variables

```
public class Customer
{
    static int count = 0;
    int id;
```

```
    Customer()
    {
        count = count + 1;
        id = count;
    }
    int getId()
    {
        return id;
    }
    public static void main(String[] args) {
        Customer c1 = new Customer();
        Customer c2 = new Customer();
        Customer c3 = new Customer();
        System.out.println("Customer 1 ID:"+c1.getId());
        System.out.println("Customer 2 ID:"+c2.getId());
        System.out.println("Customer 3 ID:"+c3.getId());
    }
}
```

Output:

```
Customer 1 ID:1
Customer 2 ID:2
Customer 3 ID:3
```

STATIC METHODS

- Static methods in Java belong to the class rather than any specific instance of the class. By declaring a method static, it becomes a class method.
- You can call static methods without creating an object of the class. These methods don't require any object state to operate.
- Since static methods are not belong to any specific object instance, they cannot directly access instance variables or methods. If you try to access instance variables or methods, you'll get a compilation error.
- Static methods can directly access other static variables and methods in the same class.
- Static methods are commonly used for utility functions, such as mathematical calculations, string manipulations, or other operations that don't require object state.

Program Example:

```
class MathUtils
{
   public static int square(int num)
   {
      return num * num;
   }
   public static int sum(int a, int b)
   {
      return a + b;
   }
}
public class StaticDemo {
   public static void main(String[] args) {
      // Calling static methods without creating an object of MathUtils
      int result1 = MathUtils.square(5);
      int result2 = MathUtils.sum(10, 20);
      System.out.println("Square of 5: " + result1);
      System.out.println("Sum of 10 and 20: " + result2);
   }
}
```

Explanation:

MathUtils Class:

The MathUtils class contains two static methods: square and sum.

StaticDemo Class:

In the Main class, the static methods square and sum are called directly on the MathUtils class without creating an instance of MathUtils.

The results are printed to the console.

METHOD OVERLOADING

In Java, Method overloading is a feature that allows a class to have more than one method with the same name,with different parameter lists. Method overloading is achieved by varying the number, type, or order of parameters in the methods.

When we call a method of an object, Java Compiler matches up the method name first and then the number and type of parameters to decide which one of overloaded methods to be invoked, this is called Compile Time Polymorphism.

Compile-Time Polymorphism: Method overloading is an example of compile-time polymorphism, where the method to be called is determined at compile time based on the method signature.

Method Signature includes the method name and arguments list but not the return type.

Return Type: The return type of overloaded methods can be different, but it does not considered to achieve method overloading.

Example of Method Overloading:

```
class MathOperations {
	public int add(int a, int b)
	{
		return a + b;
	}

	public int add(int a, int b, int c)
	{
		return a + b + c;
	}
	public double add(double a, double b)
	{
	return a + b;
	}
}
public class MethodOverloading {
	public static void main(String[] args)  {
	MathOperations obj = new MathOperations();
	System.out.println(obj.add(5, 10));
	System.out.println(obj.add(5, 10, 15));
	System.out.println(obj.add(5.5, 10.5));
	}
}
```

Explanation:

- add(int a, int b) : Adds two integers.
- add(int a, int b, int c) : Adds three integers.
- add(double a, double b) : Adds two double values.

When you call `obj.add(...)`, Java Compiler determines which method to invoke based on the number and types of arguments you pass.

CONSTRUCTORS

- Constructor is a special method, used to initialize the data members of the class at the time of its instantiation.
- A Constructor method can be invoked at the time of creating the object.
- Constructor name is similar to the name of the class and have no return type.

Constructors are of following types:

1. **Default Constructor**
2. **User – Defined Constructor**
   a) Parameterized
   b) Non – Parameterized

- **Default Constructor:** This constructor is not defined by the programmer, it is implicitly assumed by compiler that initializes the data members of the class to the standard default values at the time of creating objects when memory is allocated.

Program Example for Default Constructor:

```
class Rectangle
{
        int length;
        int breadth
        int area( )
        {
                return length * breadth;
        }
}
```

Note:- For the above class, if an object is created, then the data members of the class – length and breadth are initialized to default value 0, because no constructor is defined, therefore default constructor is invoked when the object is created.

If we wish to initialise the instance data members of the class to our own values then we have to define our own constructor **i.e. user – defined constructor.**

- **User – Defined Constructor:** User – Defined Constructor is the one that initializes the data members of the class to the user specified values, by invoking the constructor along with the arguments to be passed.
  - Therefore, defining our own constructor is meant that the default constructor is changed to our need.
  - A user defined constructor may be either parameterized and non – parameterized constructor:

Program Example for Non - Parameterized Constructor:

```
class Rectangle
{
int length;
int breadth
Rectangle()
{
        length = 10;
        breadth = 20 ;
}
int area( )
{
        return length * breadth;
}
}
```

Note:- For the above class, if an object is created, then the data members of the class – length and breadth are initialized to 10 and 20 instead of default values, because a own constructor is defined by the user.

Program Example for Parameterized Constructor:

```
class Rectangle
{
        int length;
        int breadth
Rectangle(int L,int B)
{
        length = L;
        breadth = B ;
}
int area( ) {
        return length * breadth;
}
}
```

Note:- For the above class, if an object is created, then the data members of the class – length and breadth are initialized to L and B that are passed as arguments.

CONSTRUCTOR OVERLOADING

- In Java, Constructor overloading is a concept where a class can have more than one constructor with different parameter lists.
- This allows you to create objects in different ways, depending on the parameters passed to the constructor.
- Each overloaded constructor performs a different initialization based on the arguments provided.

Program Example for Constructor Overloading:

```
class Box {
    double width, height, length;
    Box(double w, double h, double d){
        width = w;
        height = h;
        length = d;
    }
    Box(double len) {
        width = height = length = len;
    }
    double volume() {
        return width * height * length;
    }
    }
    public class ConstructorOverloading {
        public static void main(String args[]) {
            Box cuboid = new Box(10, 20, 15);
            Box cube = new Box(7);
            double vol;
            vol = cuboid.volume();
            System.out.println("Volume of Cuboid is " + vol);
            vol = cube.volume();
            System.out.println("Volume of Cube is " + vol);
        }
    }
```

Output

Volume of Cuboid is 3000.0

Volume of Cube is 343.0

PASSING ARGUMENTS BY VALUE AND BY REFERENCE

PASSING ARGUMENTS BY VALUE:

In Java, arguments are passed by value, which means that when you pass a variable to a method, a copy of the variable's value is made and passed to the method.

When you pass a variable of primitive type (e.g., int, float, char) to a method, the method receives a copy of the value. Modifying that variable inside the method does not affect the original variable.

Example Illustration:

```
public class PassByValue
{
   public static void modify(int number)
   {
      number = 10;
   }
   public static void main(String[] args)
   {
      int original = 5;
      modify(original);
      System.out.println("Original number after method call: " + original);
   }
}
```

Output:

```
Original number after method call: 5
```

PASSING ARGUMENTS BY REFERENCE (OBJECTS):

In Java, passing arguments by reference is not supported. Java only supports pass-by-value, meaning that when you pass an argument to a method, a copy of the value is made.

When you pass an object, a copy of the reference to the object is passed. The method can modify the object’s state because both the original reference and the copy reference point to the same object.

Example Illustration:

```
public class PassByReference
{
   static class MyObject
   {
      int value;
   }
```

```
    public static void modify(MyObject obj)
    {
        obj.value = 10; // This change will affect the original object
    }

    public static void main(String[] args)
    {
        MyObject obj = new MyObject();
        obj.value = 5;
        modify(obj);
        System.out.println("Object value after method call: " + obj.value);
    }
}
```

Output:

Object value after method call: 10

CLASS OBJECTS AS PARAMETERS IN METHODS

- In Java, you can pass objects as parameters to methods just like you would with primitive data types. When you pass an object as a parameter, what actually gets passed is a reference to that object, not the object itself.

Let's look at an example to illustrate the concept:

```
class Farm {
    double length;
    double width;
    Farm(double length, double width)
    {
        this.length = length;
        this.width = width;
    }
    boolean isEqual(Farm other)
    {
        if (this.length * this.width == other.length * other.width)
          return true;
        else
          return false;
    }
}
```

```
public class EqualFarm{
   public static void main(String[] args) {
      Farm farm1 = new Farm(100, 50);
      Farm farm2 = new Farm(200, 25);
      if (farm1.isEqual(farm2))
      {
         System.out.println("The areas of the two farms are equal.");
      } // if close
      else
      {
         System.out.println("The areas of the two farms are not equal.");
      } // else close
   } // main close
} // class close
```

Further, if you modify the object within the method, the changes will be reflected outside the method as well.

/* Write an example Program given in the concept Passing arguments by reference here*/

RECURSIVE METHODS

- Recursion is a technique where a method calls itself directly or indirectly to solve a problem.
- Recursion is a common approach in programming for solving problems that can be broken down into smaller, similar subproblems.
- Every recursive solution has two major cases:
  - **Base Case:** The simplest instance of the problem, which can be solved directly without further recursion. This stops the recursion.
  - **Recursive Case:** The part of the method that breaks the problem down into smaller subproblems and calls the method on those smaller instances.

The following are the rules for designing a recursive function:

1. First determine the base case
2. Then determine the recursive case
3. Finally combine the base case and recursive case into a function

To understand the recursive functions, take the classic example of calculating the factorial of a number. The factorial of the number is the product of the integral values from 1 to the number.

This means,

n! = n * (n – 1)!

Here, n! is obtained by multiplying **n** with **(n – 1)!**

Assume n=3 so 3! = 3 x 2 x 1

This can be written as

3! = 3 x 2!, where 2! = 2 x 1!

So, 3! = 3 x 2 x 1!, where 1! = 1

So, 3! = 3 x 2 x 1.

If one should observe the problem carefully, here the product is the repetitive task. So it is suitable to write a recursive function. As every recursive function has a base case and recursive case, they are defined a follows for factorial function:

- **Base Case** is when n =l will be 1 as 1! = 1
- **Recursive Case** is the factorial function itself but with a value of **n,** this can be given as

  Factorial (n) = n x Factorial (n – 1)

The following program calculates the factorial of a number recursively:

```
public class Factorial
{
	public static int factorial(int n)
	{
	if (n == 0)
	{
		return 1;
	}
	else
	{
		return n * factorial(n - 1);
	}
	}

	public static void main(String[] args)
	{
	int number = 5;
	int result = factorial(number);
	System.out.println("Factorial of " + number + " is: " + result);
	}
}
```

Output:

Factorial of 5 is 120

Fibonacci Numbers:

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence typically looks like this: 0, 1, 1, 2, 3, 5, 8, 13, ...

Here's a Java program that calculates and prints the Fibonacci series up to a given number of terms using both recursive and iterative approaches:

```
public class Fibonacci
{
   public static int fib(int n)
   {
      if (n == 1)
        return 0;
      else if(n == 2)
        return 1;
      return fib(n - 1) + fib(n - 2);
   }

   public static void main(String[] args)
   {
   Fibonacci obj = new Fibonacci();
   for(int i=1;i<=10;i++ )
   {
        int term=obj.fib(i);
        System.out.println(term);
   }
   }
}
```

Output:

```
0       1       1       2       3       5       8       13      21      34
```

NESTING OF METHODS

In Java, "nesting of methods" typically refers to the concept where a method is called within another method of same class. In more detail, calling a method from another method is called nesting of methods but one method of a class can call only other methods of same class but not the methods of another class.

This is a common practice in object-oriented programming, allowing you

- to organize your code better
- promote code reuse
- improve readability.

When one method calls another, the called method executes and returns a result, after which the calling method can continue execution with the returned result.

Here's a simple example to demonstrate nesting of methods:

```
public class Calculator
{
   public int add(int a, int b)
  {
      return a + b;
   }
   public int subtract(int a, int b)
   {
      return a - b;
   }
   public int conditionalCompute(int a, int b, boolean addOperation) {
      if (addOperation) {
         return add(a, b); // Nested method call
      } else {
         return subtract(a, b); // Nested method call
      }
   }
   public static void main(String[] args) {
      Calculator calc = new Calculator();
      int x = calc.conditionalCompute(10, 5, true)
      System.out.println("Result (Addition): " + x );
      x = calc.conditionalCompute(10, 5, false);
      System.out.println("Result (Subtraction): " + x );
   }
}
```

Output:

```
Result (Addition): 15
Result (Subtraction): 5
```

In this example, the conditionalCompute method calls either add or subtract based on the boolean value passed to it.

OVERRIDING METHODS

A method defined in super class can be inherited by its sub class. This feature of inheritance allows us to define and use the same methods as in the super class. Inheritance allows deriving new classes from the old one and reusing the properties of the existing one.

In some cases, it may be required that objects in a sub class use the same method that is defined in the super class with a different behaviour. Then, in that case, the method is said to be overridden. The method defined in sub class will be executed and it will hide the definition given in the super class.

The super class definition can also be executed if it is used with attribute super.

Method overridding is different from method overloading, in which several methods have the same name but the parameter list has to be different in type or number or order of parameters. But in case of method overridding, the method signatures defined in the super class and sub class have been exactly same.

ATTRIBUTE final

In Java, The keyword final 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

Final variables: When a variable is declared as final, its value cannot be changed once it has been initialized.

Example Program:

```
class Test
{
	final int minmarks = 35;
	void change()
	{
		minmarks = 40; // error: cannot assign a value to final variable
	}
	public static void main(String[] args)   {
		Test t = new Test();
		t.change();
		System.out.println("Min Marks:"+t.minmarks);
	}
}
```

Final methods: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that should not be allowed to modify by subclasses.

Example Program:

```
class Motor{
        final void display() {
        System.out.println("Ready");
        }
}
class ACMotor extends Motor{
        void display() {
        System.out.println("Unable to Start");
        }
}
```

The above program gives the following error:

error: display() in ACMotor cannot override display() in Motor

Final classes: When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as it is and should not be modified or extended.

Example:

- ***Below is the declaration of final class Vehicle***

```
final class Vehicle
{
   public void displayType()
   {
      System.out.println("This is a vehicle.");
   }
 }
```

- ***If the above final class is inherited like below:***

```
class Car extends Vehicle
{


}
```

This would cause a compile-time error.

```
error: cannot inherit from final Vehicle
class Car extends Vehicle {
^
1 error
```

NESTED CLASSES

- Nested classes are those classes defined within another class or interface.
- Nested class exists only as along as the enveloping class is existed. Therefore, the scope of the nested class is limited to the enveloping class.
- They help group classes that belong together, making code more readable and logical.
- There are four types of nested classes:
  1. Member Inner Class or Non – static Inner Class
  2. Anonymous Class
  3. Local Class
  4. Static Nested Class

I. Member Inner Class:

- A member inner class, also known as a non-static inner class
- It is a type of nested class that is defined within another class It is outside any method, constructor, or block.
- It has access to the outer class's instance variables and methods, including private members.
- To create an instance/object of the inner class, you first need an instance of the outer class.
- The inner class can access both static and non-static members of the outer class.
- Member inner classes are useful when you need to encapsulate functionality that is tightly bound to an instance of the outer class.

Example:

Here’s an example to illustrate how a member inner class works:

```
class OuterClass
{
   private String s = "Outer class variable";
   class InnerClass      // Member inner class
   {
      void display()
      {
        System.out.println("Accessing: " + s);
      } // display close
   } // inner class close
} // outer class close
```

```
public class MemberNestedClass
{
   public static void main(String[] args)
   {
   OuterClass outerObject = new OuterClass(); // Object of Outer Class
   OuterClass.InnerClass innerObject = outerObject.new InnerClass(); // Object of inner class
   innerObject.display();
   }
}
```

Explanation

- OuterClass contains a private instance variable S and a member inner class InnerClass.
- InnerClass has a method display() that accesses outerVar from the enclosing OuterClass.
- In the main method:
  - An instance of OuterClass is created.
  - Using this instance, an instance of InnerClass is created.
  - The display() method of InnerClass is called, which prints the value of s.

The non-staic inner class can access all members (including private) of the outer class because it's logically connected to an instance of the outer class.

II. Anonymous Class

- An anonymous class is a type of inner class without a name, which is used to instantiate objects on the fly.
- Anonymous classes are typically used to implement interfaces or extend classes.
- They are defined and instantiated in a single expression.
- They are generally used for short-term tasks where defining a separate class would be excessive.
- The anonymous class is defined within a single expression, making it ideal for concise and short-lived tasks.

Example Program:

```
class Animal
{
   void sound()
   {
      System.out.println("Animal makes a sound");
   }
}
```

```
public class AnonymousClass
{
   public static void main(String[] args)
   {
      // Anonymous class extending the Animal class
      Animal dog = new Animal() {
         @Override
         void sound() {
            System.out.println("Dog barks");
         }
       };
      // Calling the overridden method
      dog.sound();
   }
}
```

In the above example, the anonymous class is defined and instantiated using the new keyword followed by the class name.

superclass methods are directly overridden within the anonymous class body.

III. Local Class

- A local class is a type of inner class that is defined within a block, such as a method, constructor, or even an initializer block.
- Local Classes can access local variables and parameters of the enclosing block,which are declared as final
- The visibility of a local class is restricted to the block in which it is defined.

Example:

Here's a simple example of a local class within a method:

```
class OuterClass {
   void outerMethod()
   {
      int localVar = 10;
      class LocalClass
      {
         void display()
         {
         System.out.println("Local variable: " + localVar);
         }
      } // Local Class Close
      LocalClass localObject = new LocalClass();
```

```
        localObject.display();
    } // outerMethod Close
}//Outer Class Close

public class LocalClass  {
    public static void main(String[] args) {
        OuterClass outerObject = new OuterClass();
        outerObject.outerMethod();
    }
}
```

Explanation: In the above example:

- The LocalClass is declared inside the outerMethod of OuterClass.
- LocalClass can access the local variable localVar from outerMethod
- The instance of the local class is created and used within the same block, maintaining the encapsulation.

Local classes are a powerful feature that allow you to define classes with a very limited scope, making them ideal for situations where you need to encapsulate specific functionality within a method, constructor, or block.

IV. Static Nested Class

- A static nested class is a nested class that is declared with the static modifier.
- Since it is a static member, you can create an instance of a static nested class without creating an instance of the outer class.
- A static nested class cannot directly access non-static members (instance variables and methods) of the outer class. It can only access static members of the outer class.
- A static nested class can be used to group classes that are only relevant to the outer class.

Example for static nested class:

```
class OuterClass
{
   static int x = 10;
   int y = 20;
   // Static nested class
   static class InnerClass
   {
      void display()
      {
      System.out.println("Outer class static variable: " + x);
     //System.out.println("Outer class instance variable: " + y); // This would cause a compile-time error
      }
   }
}
public class StaticNestedClass
{
   public static void main(String[] args)
   {
        OuterClass.InnerClass obj = new OuterClass.InnerClass();
        obj.display();
   }
}
```

In the above example, StaticNestedClass can access the static variable **x** of OuterClass but cannot directly access the instance variable **y**

Static nested classes are often used as helper classes that are closely related to the outer class but don’t require access to the outer class’s instance variables.