

1. Java Basics:-

1. What is Java? Explain its features.

Java is a high-level, object-oriented, platform-independent programming language developed by Sun Microsystems. It allows developers to write code once and run it anywhere using the Java Virtual Machine (JVM).

Features of Java (Short)

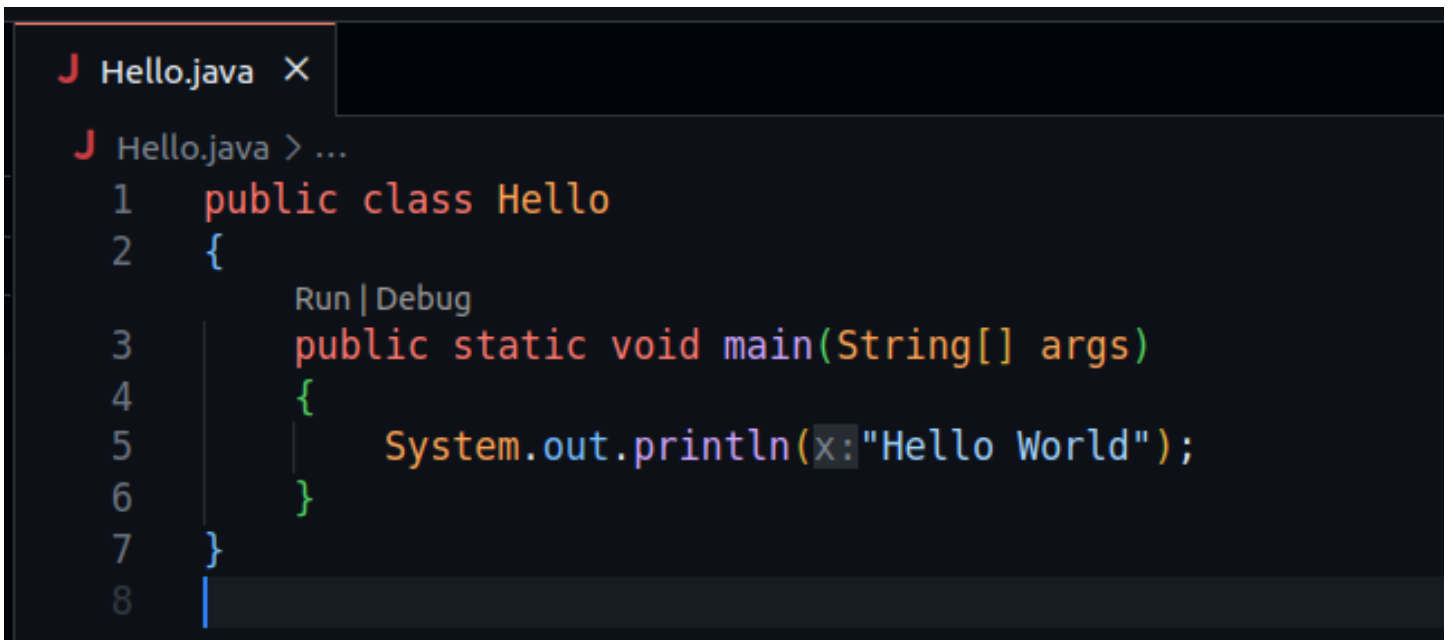
- a. **Simple** – Easy to learn, clean syntax.
- b. **Object-Oriented** – Based on OOP principles.
- c. **Platform-Independent** – Runs on any OS with JVM.
- d. **Secure** – No pointers, strong security features.
- e. **Robust** – Strong memory management & error handling.
- f. **Multithreaded** – Supports concurrent execution.
- g. **High Performance** – JIT compiler boosts speed.
- h. **Distributed** – Supports networked applications.
- i. **Dynamic** – Loads classes at runtime.
- j. **Architecture Neutral** – Not tied to specific hardware.

2. Explain the Java program execution process.

- a. Write code in a .java file
- b. Compile the code using javac to get a .class bytecode file
- c. ClassLoader loads the .class file into memory
- d. Bytecode Verifier checks code safety
- e. JVM interprets or compiles the bytecode and runs the program

This allows Java programs to run on any system with a Java Virtual Machine.

3. Write a simple Java program to display 'Hello World'.



```
J Hello.java X
J Hello.java > ...
1  public class Hello
2  {
    Run | Debug
3      public static void main(String[] args)
4      {
5          System.out.println('Hello World');
6      }
7  }
8  |
```

4. What are data types in Java? List and explain them.

Data types are divided into two groups:

- Primitive data types - includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter)

byte : Stores whole numbers from -128 to 127

short : Stores whole numbers from -32,768 to 32,767

int Stores: whole numbers from -2,147,483,648 to 2,147,483,647

long : Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

float: Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits

double: Stores fractional numbers. Sufficient for storing 15 to 16 decimal digits

boolean: Stores true or false values

char : Stores a single character/letter or ASCII values

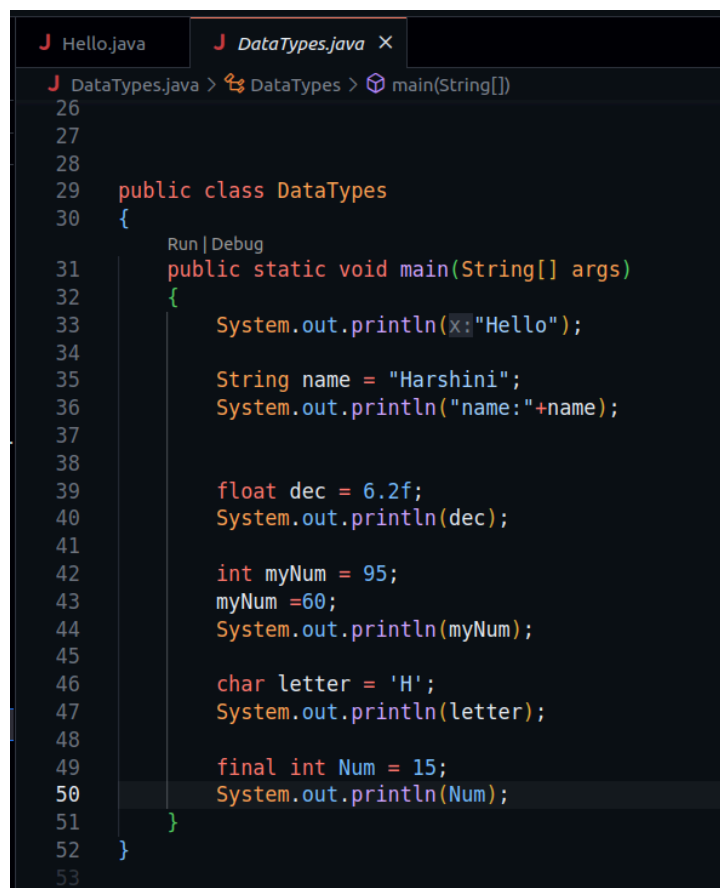
String - stores text, such as "Hello". String values are surrounded by double quotes 'h'

int - stores integers (whole numbers), without decimals, such as 123 or -123

float - stores floating point numbers, with decimals, such as 19.99 or -19.99

char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes

boolean - stores values with two states: true or false



```
J Hello.java J DataTypes.java X
J DataTypes.java > DataTypes > main(String[])
26
27
28
29 public class DataTypes
30 {
31     Run | Debug
32     public static void main(String[] args)
33     {
34         System.out.println(x: "Hello");
35
36         String name = "Harshini";
37         System.out.println("name:" + name);
38
39         float dec = 6.2f;
40         System.out.println(dec);
41
42         int myNum = 95;
43         myNum = 60;
44         System.out.println(myNum);
45
46         char letter = 'H';
47         System.out.println(letter);
48
49         final int Num = 15;
50         System.out.println(Num);
51     }
52 }
53
```

5. What is the difference between JDK, JRE, and JVM?

JDK (Java Development Kit)

It is the full package for Java development. It includes JRE, JVM, and development tools like the compiler.

JRE (Java Runtime Environment)

It provides the environment to run Java programs. It includes JVM and libraries but **not** development tools.

JVM (Java Virtual Machine)

It is the engine that runs Java bytecode. It is part of both JDK and JRE.

6. What are variables in Java? Explain with examples.

Variables in Java are containers used to store data values. Each variable has a type, a name, and can hold a value.

```
53
54
55 public class Variable {
56     int age = 25;           // instance variable
57     static String name = "Raj"; // static variable
58
59     public void display() {
60         int marks = 90;     // local variable
61         System.out.println(marks);
62     }
63 }
64
```

In this example:

- age is an instance variable
- name is a static variable
- marks is a local variable

7. What are the different types of operators in Java?

Java provides several types of operators to perform various operations on variables and values.

1. Arithmetic Operators

Used to perform basic mathematical operations.

Operator	Description	Example
+	Addition	$a + b$
-	Subtraction	$a - b$
*	Multiplication	$a * b$
/	Division	a / b
%	Modulus (Remainder)	$a \% b$

2. Relational (Comparison) Operators

Used to compare two values and return a boolean (true or false).

Operator	Description	Example
==	Equal to	$a == b$
!=	Not equal to	$a != b$
>	Greater than	$a > b$
<	Less than	$a < b$
>=	Greater than or equal to	$a >= b$
<=	Less than or equal to	$a <= b$

3. Logical Operators

Used to combine multiple boolean conditions.

Operator	Description	Example
&&	Logical AND	(a > b) && (a > c)
,		,
!	Logical NOT	!(a > b)

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	a = 10
+=	Add and assign	a += 5 → a = a + 5
-=	Subtract and assign	a -= 5
*=	Multiply and assign	a *= 5
/=	Divide and assign	a /= 5
%=	Modulus and assign	a %= 5

5. Unary Operators

Operate on a single operand.

Operator	Description	Example
+	Unary plus	+a
-	Unary minus	-a
++	Increment	a++, ++a
--	Decrement	a--, --a
!	Logical NOT	!true

6. Bitwise Operators

Operate on bits and perform bit-level operations.

Operator	Description	Example
&	Bitwise AND	a & b
 	Bitwise OR	a b
^	Bitwise XOR	a ^ b
~	Bitwise Complement	~a
<<	Left Shift	a << 2
>>	Right Shift	a >> 2
>>>	Zero Fill Right Shift	a >>> 2

7. Ternary Operator

A shortcut for if-else statements.

Operator	Description	Example
? :	Conditional operator	result = (a > b) ? a : b;

8. instanceof Operator

Checks whether an object is an instance of a specific class.

Operator	Description	Example
instanceof	Checks object type	obj instanceof String

8. Explain control statements in Java (if, if-else, switch).

1. if Statement

Executes a block of code only if a specified condition is true.

2. if-else Statement

Executes one block of code if the condition is true, otherwise executes another block.

4. switch Statement

Used to select one option from multiple choices based on the value of a variable. Each option is called a "case".

```
J IfElse.java > IfElse > main(String[])
1  public class IfElse {
    Run | Debug
2      public static void main(String[] args) {
3          //if-else statement
4          int e = 50;
5          int f = 40;
6
7          if(e>f){
8              System.out.println(x:"e is greater than f");
9          }
10         else{
11             System.out.println(x:"f is greater than e");
12         }
13     }
14 }
15
```

```

J SwitchCase.java > SwitchCase
1 public class SwitchCase {
  Run | Debug
2 public static void main(String[] args){
3 // SwitchCase
4 int day = 4;
5 switch (day) {
6 case 1:
7     System.out.println(x:"Monday");
8     break;
9 case 2:
10    System.out.println(x:"Tuesday");
11    break;
12 case 3:
13    System.out.println(x:"Wednesday");
14    break;
15 case 4:
16    System.out.println(x:"Thursday");
17    break;
18 case 5:
19    System.out.println(x:"Friday");
20    break;
21 case 6:
22    System.out.println(x:"Saturday");
23    break;
24 case 7:
25    System.out.println(x:"Sunday");
26    break;
27 default:
28    System.out.println(x:"Something went wrong");
29 }
30 }
31 }
32

```

9. Write a Java program to find whether a number is even or odd.

```

J EvenOdd.java > ...
1 public class EvenOdd {
  Run | Debug
2 public static void main(String[] args) {
3     int number = 50;
4     if (number % 2 == 0) {
5         System.out.println(number + " is Even");
6     } else {
7         System.out.println(number + " is Odd");
8     }
9 }
10 }

```

```
● harshini@harshini-Inspiron-3505:~/Desktop/demo$ ./utilsInExceptionMessages -cp /home/harshini/.config/Cod
4e/bin EvenOdd
50 is Even
○ harshini@harshini-Inspiron-3505:~/Desktop/demo$
```

10. What is the difference between while and do-while loop?

Difference Between while and do-while Loop in Java

- In a while loop, the condition is checked before the loop body is executed.
- In a do-while loop, the condition is checked after the loop body is executed.
- The while loop may not run at all if the condition is false initially.
- The do-while loop will always run at least once, even if the condition is false.
- while loop is used when you want to run the loop only if the condition is true.
- do-while loop is used when you want the loop to run once before checking the condition.

```
J WhileLoop.java > WhileLoop > main(String[])
1  public class WhileLoop {
    Run | Debug
2      public static void main(String[] args) {
3          //While loop
4          int i = 0;
5          while (i <= 10) {
6              System.out.println(i);
7              i++;
8          }
9      }
10
11 }
12
```

J Dowhile.java > ...

```
1  public class Dowhile {
    Run | Debug
2      public static void main(String[] args) {
3          //do while
4          int k = 0;
5          do {
6              System.out.println(k);
7              k++;
8          }while (k <= 5);
9      }
10 }
11
```

2. Object-Oriented Programming (OOPs)

1. What are the main principles of OOPs in Java? Explain each.

Java is based on the Object-Oriented Programming (OOP) model. The four main principles of OOP are:

a. **Encapsulation**

- It is the process of wrapping data (variables) and code (methods) into a single unit (class).
- It protects data by making variables private and accessing them through public methods (getters/setters).

b. **Inheritance**

- It allows a class (child) to inherit properties and methods from another class (parent).
- It promotes code reusability and method overriding.

c. **Polymorphism**

- It means “many forms.”
- A single method or object can behave differently in different situations.
- It is achieved through method overloading (compile-time) and method overriding (runtime).

d. **Abstraction**

- It hides complex implementation details and shows only the essential features.

- Achieved using abstract classes and interfaces in Java.

2. What is a class and an object in Java? Give examples.

Class and Object in Java

Class:

- A class is a blueprint or template for creating objects.
- It defines properties (variables) and behaviors (methods) of objects.

Object:

- An object is an instance of a class.
- It represents a real-world entity with state and behavior.

```
J ClassesAndObjects.java > ...
1  public class ClassesAndObjects {
2      public void display() {
3          System.out.println(x:"Hello, this is a method in ClassObj.");
4      }
5      public void show() {
6          System.out.println(x:"This is show method in ClassObj.");
7      }
8      Run | Debug
9      public static void main(String[] args) {
10         //create an instance of ClassObj
11         ClassesAndObjects obj = new ClassesAndObjects();
12         obj.display();
13         obj.show();
14     }
15 }
16
```

3. Write a program using class and object to calculate area of a rectangle.

```
J Rectangle.java > ...
1  public class Rectangle {
2      int length;
3      int width;
4
5      void Area() {
6          int area = length * width;
7          System.out.println("Area of Rectangle: " + area);
8      }
   Run | Debug
9  public static void main(String[] args) {
10     Rectangle rect = new Rectangle();
11     rect.length = 10;
12     rect.width = 5;
13     rect.Area();
14 }
15 }
16
```

```
● harshini@harshini-Inspiron-3505:~/Desktop/demo$
Area of Rectangle: 50
○ harshini@harshini-Inspiron-3505:~/Desktop/demo$
```

4.Explain inheritance with real-life example and Java code.

Inheritance is an OOP principle where one class (child) inherits the properties and behaviors of another class (parent). It promotes **code reusability** and supports **method overriding**.

```

J SingleInheritance.java > ...
1 // Here Animal is Parent class where as Cat is child class.
2 // Child class can use the properties of parent class but parent class can't use the methods and functions and properties of childclass
3 // This is a example of Single inheritance as there is a single parent and single child.
4
5
6 class Animal{
7     void eat(){
8         System.out.println(x:"Animal is eating");
9     }
10 }
11
12 class Cat extends Animal{
13     void meow(){
14         System.out.println(x:"cat is meowing");
15     }
16 }
17
18 public class SingleInheritance {
19
20     Run | Debug
21     public static void main(String[] args) {
22         Cat c = new Cat();
23         c.meow();
24         c.eat();
25
26         Animal a= new Animal();
27         a.eat();
28     }
29 }

```

```

J MultiLevelInheritance.java > ...
1 // here the class Animal is Grand parent class.
2 // Cat is Parent class and the child class of Animal, can access the properties of its parent class Animal only.
3 // Lastly Kitten is Child class of Cat and this class can access properties and functions of both the parent classes.
4 class Animal{
5     void ani(){
6         System.out.println(x:"HI there, I am an Animal!!");
7     }
8 }
9 class Cat extends Animal{
10     void cat(){
11         System.out.println(x:"HI there, I am an Cat!!");
12     }
13 }
14 class Kitten extends Cat{
15     void kit(){
16         System.out.println(x:"HI there, I am an Kitten!!");
17     }
18 }
19 public class MultiLevelInheritance {
20     Run | Debug
21     public static void main(String[] args) {
22         // Animal class can perform its own functions and methods only.
23         Animal a = new Animal();
24         a.ani();
25         // Cat class can perform its own functions and methods as well as the functions and methods of Animal class.
26         Cat c = new Cat();
27         c.ani();
28         c.cat();
29         // Kitten class can perform its own functions and methods as well as the functions and methods of Animal class and Cat class.
30         Kitten k = new Kitten();
31         k.ani();
32         k.cat();
33         k.kit();
34     }
35 }

```

```

J MultipleInheritance.java > ...
1 // implements keyword used if we are using the relation between: CI(class and Interface) and IC(Interface and Class).
2 // For CC( Class and Class) and II(Interface and Interface) use extends
3 interface p1{
4     default void parent1(){
5         System.out.println(x:"Hello there, I am Parent 1 of child class.");
6     }
7 }
8 interface p2{
9     default void parent2(){
10        System.out.println(x:"Hello there, I am Parent 2 of child class.");
11    }
12 }
13 class childClass implements p1, p2{
14     void hello(){
15         System.out.println(x:"Hello there, I am the Child Class.");
16     }
17 }
18 public class MultipleInheritance {
19     Run | Debug
20     public static void main(String[] args) {
21         childClass c = new childClass();
22         c.hello();
23         c.parent1();
24         c.parent2();
25     }
26 }

```

```

J HierarchicalInheritance.java > ...
1 // its not compulsory that the main class should be in the the public class it can be in any either of the classes also.
2 class Animal {
3     void eat() {
4         System.out.println(x:"Animal is eating");
5     }
6 }
7 class Cat extends Animal {
8     void meow() {
9         System.out.println(x:"cat is meowing");
10    }
11 }
12 class Dog extends Animal {
13     void bark() {
14         System.out.println(x:"dog is barking");
15    }
16 }
17 public class HierarchicalInheritance {
18     Run | Debug
19     public static void main(String[] args) {
20         Cat c = new Cat();
21         c.meow();
22         c.eat();
23
24         Animal a = new Animal();
25         a.eat();
26
27         Dog d = new Dog();
28         d.bark();
29         d.eat();
30     }
31 }

```



```

J HybridInheritance.java > HybridInheritance
9  // Interface 1
10 interface Printable {
11     void print();
12 }
13 // Interface 2
14 interface Showable {
15     void show();
16 }
17 // Base class
18 class Animal {
19     void eat() {
20         System.out.println(x:"This animal eats food.");
21     }
22 }
23 // Derived class 1
24 class Dog extends Animal {
25     void bark() {
26         System.out.println(x:"Dog barks.");
27     }
28 }
29 // Final class implementing multiple interfaces
30 class HybridDog extends Dog implements Printable, Showable {
31     public void print() {
32         System.out.println(x:"Printing from Printable interface.");
33     }
34     public void show() {
35         System.out.println(x:"Showing from Showable interface.");
36     }
37     void displayInfo() {
38         eat(); // from Animal
39         bark(); // from Dog
40         print(); // from Printable
41         show(); // from Showable
42     }
43 }
44 // Main class to run the program
45 public class HybridInheritance {
46     public static void main(String[] args) {
47         HybridDog hd = new HybridDog();
48         hd.displayInfo();
49     }
50 }

```

5. What is polymorphism? Explain with compile-time and runtime examples.

Polymorphism means "**many forms**". In Java, it allows one interface, method, or object to behave in different ways. It is a key concept of Object-Oriented Programming (OOP).

There are **two types** of polymorphism in Java:

1. Compile-time Polymorphism (Method Overloading)

- It occurs when **multiple methods** in the same class have **the same name but different parameters**.
- The method to be called is decided **at compile time**.

J OverloadingMethod.java > ...

```
1  class Addition{
2      public int add(int a, int b){
3          return a+b;
4      }
5      public int add(int a, int b, int c){
6          return a+b+c;
7      }
8      public double add(double a, double b){
9          return a+b;
10     }
11 }

12
13 public class OverloadingMethod {
14     Run | Debug
15     public static void main(String[] args) {
16         Addition x = new Addition();
17         int ans1 = x.add(a:1, b:5);
18         int ans2 = x.add(a:9, b:7, c:5);
19         double ans3 = x.add(a:5.54, b:55.5);
20
21         System.out.println("Answer 1 is: "+ans1);
22         System.out.println("Answer 2 is: "+ans2);
23         System.out.println("Answer 3 is: "+ans3);
24     }
25 }
26
```

2. Runtime Polymorphism (Method Overriding)

- It occurs when a **subclass overrides a method** of its superclass.
- The method to be called is decided **at runtime** using the object type.

```

Polymorphism.java > Polymorphism
7
8  class Animal {
9      void sound() {
10         System.out.println(x:"Animal is speaking");
11     }
12 }
13
14 class Cat extends Animal {
15     void sound() {
16         super.sound();
17         System.out.println(x:"cat is meowing");
18     }
19 }
20
21 class Dog extends Animal {
22     void sound() {
23         super.sound();
24         System.out.println(x:"dog is barking");
25     }
26 }
27
28 public class Polymorphism {
29     Run | Debug
30     public static void main(String[] args) {
31         Animal d = new Dog();
32         d.sound();
33         Cat c = new Cat();
34         c.sound();
35     }
36 }

```

6. What is method overloading and method overriding? Show with examples.

1. Method Overloading (Compile-time Polymorphism)

- **Same method name, different parameters** (type or number).
- Happens **within the same class**.
- Resolved **at compile time**.

2. Method Overriding (Runtime Polymorphism)

- **Same method name and parameters, but different implementation** in child class.
- Happens **between parent and child class**.
- Resolved **at runtime**.

J OverLoadingExample.java > ...

```
1 public class OverLoadingExample{
2     // Method with two int parameters
3     void add(int a, int b) {
4         System.out.println("Sum of integers: " + (a + b));
5     }
6
7     // Method with two double parameters
8     void add(double a, double b) {
9         System.out.println("Sum of doubles: " + (a + b));
10    }
11
12    // Method with three parameters
13    void add(int a, int b, int c) {
14        System.out.println("Sum of three integers: " + (a + b + c));
15    }
16
17    Run | Debug
18    public static void main(String[] args) {
19        OverLoadingExample obj = new OverLoadingExample();
20        obj.add(a:5, b:10);           // Calls method with int, int
21        obj.add(a:3.5, b:2.5);       // Calls method with double, double
22        obj.add(a:1, b:2, c:3);       // Calls method with three ints
23    }
```

J OverridingExample.java > ...

```
1 // Parent class
2 class Animal {
3     void makeSound() {
4         System.out.println(x:"Animal makes a sound");
5     }
6 }
7
8 // Child class
9 class Cat extends Animal {
10     @Override
11     void makeSound() {
12         System.out.println(x:"Cat meows");
13     }
14 }
15
16 // Main class
17 public class OverridingExample {
18     Run | Debug
19     public static void main(String[] args) {
20         Animal a = new Animal(); // Base class reference and object
21         Animal b = new Cat();     // Base class reference to child object
22
23         a.makeSound(); // Calls Animal version
24         b.makeSound(); // Calls Cat version (overridden)
25     }
```

7. What is encapsulation? Write a program demonstrating encapsulation.

Encapsulation is the process of binding data (variables) and code (methods) into a single unit, typically a class, and restricting direct access to some of the object's components. It is achieved by:

- Making variables private
- Providing public getter and setter methods to access and modify those variables

Benefits of Encapsulation:

- Protects data from unauthorized access
- Increases code maintainability and flexibility
- Makes the class easier to use and modify

```
J Encapsulation.java > Encapsulation > setAge(int)
1  //Encapsulation in java
2  public class Encapsulation {
3      // Step 1: Private data members
4      private String name;
5      private int age;
6
7      // Step 2: Public getter method for name
8      public String getName() {
9          return name;
10     }
11
12     // Step 3: Public setter method for name
13     public void setName(String name) {
14         this.name = name;
15     }
16
17     // Getter method for age
18     public int getAge() {
19         return age;
20     }
21
22     // Setter method for age
23     public void setAge(int age) {
24         if (age > 0) { // simple validation
25             this.age = age;
26         }
27     }
28
29     Run | Debug
30     public static void main(String[] args) {
31         Encapsulation s = new Encapsulation();
32         s.setName(name:"Harshada");
33         s.setAge(age:20);
34
35         System.out.println("Student Name: " + s.getName());
36         System.out.println("Student Age: " + s.getAge());
37     }
38 }
```

8. What is abstraction in Java? How is it achieved?

Abstraction is one of the main principles of Object-Oriented Programming (OOP). It means **hiding the internal details** and showing only the **essential features** to the user.

Abstraction in Java is achieved using:

a. **Abstract Classes**

- A class declared with the abstract keyword.
- It can have both **abstract methods** (no body) and **concrete methods** (with body).
- Cannot be instantiated directly.

b. **Interfaces**

- An interface is a collection of abstract methods (by default).
- A class implements an interface to provide method definitions.
- Supports full abstraction (100%).

```
// Abstract class
abstract class Animal {

    // Abstract method (no body)
    abstract void makeSound();

    // Regular method (has body)
    void eat() {
        System.out.println(x:"This animal eats food.");
    }
}

// Subclass that extends the abstract class
class Dog extends Animal {

    // Implementing the abstract method
    void makeSound() {
        System.out.println(x:"Bark!");
    }
}

// Main class to run the code
public class Abstraction {
    Run | Debug
    public static void main(String[] args) {
        Dog d = new Dog(); // Create a Dog object
        d.makeSound();      // Calls Dog's version of makeSound()
        d.eat();            // Calls method from abstract class
    }
}
```

```

// Interface (fully abstract)
interface Animal {

    // Abstract method
    void makeSound();
}

// Class that implements the interface
class Cat implements Animal {

    // Implementing the interface method
    public void makeSound() {
        System.out.println(x:"Meow!");
    }
}

// Main class to run the code
public class Abstraction {
    Run | Debug
    public static void main(String[] args) {
        Cat c = new Cat(); // Create Cat object
        c.makeSound();      // Calls method defined in Cat
    }
}

```

9. Explain the difference between abstract class and interface.

An abstract class can have both abstract and non-abstract methods, while an interface only has abstract methods (until Java 7), and from Java 8 onward, it can have default and static methods.

Abstract classes are declared using the abstract keyword; interfaces are declared using the interface keyword.

Abstract classes can have constructors; interfaces cannot have constructors.

Abstract classes can have instance variables; interfaces can only have public static final (constants).

Abstract classes support single inheritance; interfaces support multiple inheritance (a class can implement multiple interfaces).

Methods in abstract classes can have any access modifier (public, protected, private); all methods in interfaces are public by default.

Use an abstract class when classes are closely related and share common code.

Use an interface to define a common behavior for unrelated classes.

Abstract classes provide partial abstraction; interfaces provide full abstraction (100%).

Abstract class is ideal when you want to define a base class with some shared code, while interfaces are best for defining a contract without implementation.

10. Create a Java program to demonstrate the use of interface.

```
// Interface (fully abstract)
interface Animal {

    // Abstract method
    void makeSound();
}

// Class that implements the interface
class Cat implements Animal {

    // Implementing the interface method
    public void makeSound() {
        System.out.println(x:"Meow!");
    }
}

// Main class to run the code
public class Abstraction {
    Run|Debug
    public static void main(String[] args) {
        Cat c = new Cat(); // Create Cat object
        c.makeSound();      // Calls method defined in Cat
    }
}
```