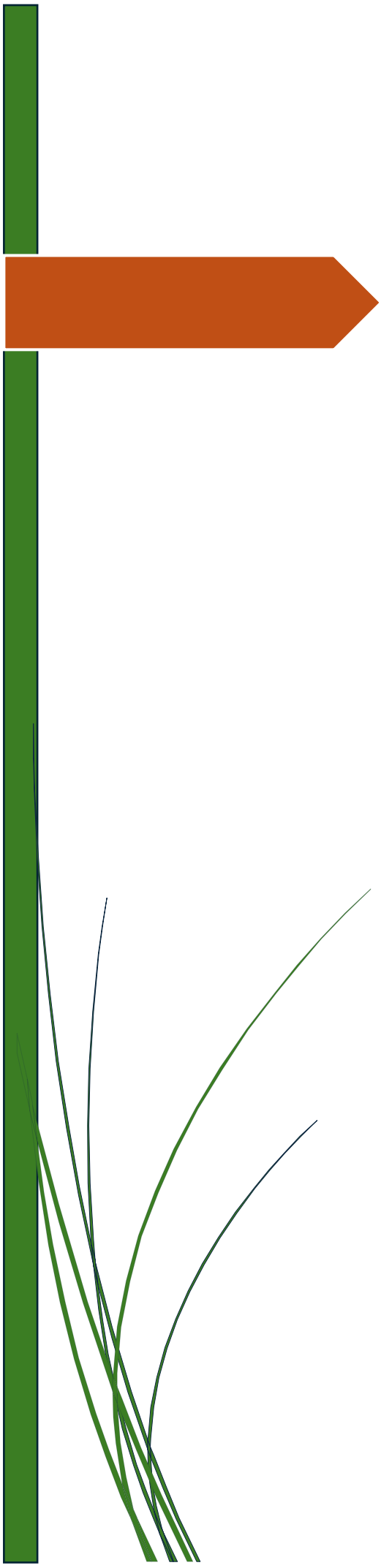
Programming in Java

GATE Computer Education



PROGRAMMING IN JAVA

Table of Contents

1. Introduction to Java

Java is a high-level, object-oriented programming language developed by Sun Microsystems in the mid-1990s. It has become one of the most popular programming languages due to its versatility, portability, and robustness.

Key Features of Java

1. Platform Independence
- Write Once, Run Anywhere (WORA): Java code is compiled into bytecode, which can be executed on any platform that has a Java Virtual Machine (JVM). This makes Java programs highly portable across dif erent operating systems.

2. Object-Oriented
- Encapsulation: Bundling data and methods that operate on the data within one unit (class), and restricting access to some of the object's components.
- Inheritance: Mechanism where one class acquires the properties and behaviors of another class.
- Polymorphism: Ability to process objects dif erently based on their data type or class.
- Abstraction: Hiding complex implementation details and showing only the essential features of an object.

3. Simple and Easy to Learn
- Java's syntax is straightforward and closely resembles C++, but with signif cant improvements to eliminate complexities and potential errors.

4. Robust
- Exception Handling: Java provides a robust exception handling mechanism that helps manage runtime errors.
- Memory Management: Automatic garbage collection helps manage memory allocation and deallocation, reducing the risk of memory leaks.

5. Secure
- Java provides several features to ensure security, including the sandbox environment of the JVM, which prevents unauthorized access to system resources.

6. Multithreaded
- Java supports multithreading, allowing concurrent execution of two or more threads. This helps in making ef icient use of CPU resources.

7. Network-Centric
- Java has built-in support for networking, making it easy to create applications that communicate over the internet.

8. Rich Standard Library
- Java comes with a comprehensive standard library (Java API) that includes a wide range of pre-written classes and methods to perform common tasks.

Basic Java Components

1. Java Development Kit (JDK)

- JDK: Includes tools for developing and running Java programs. It contains the Java Compiler (javac), Java Runtime Environment (JRE), and other tools.

2. Java Runtime Environment (JRE)
- JRE: Provides the libraries, Java Virtual Machine (JVM), and other components necessary to run Java applications.

3. Java Virtual Machine (JVM)
- JVM: Executes Java bytecode and provides a platform-independent execution environment. It translates bytecode into machine code specif c to the operating system.

Basic Java Syntax

1. Classes and Objects
- A class is a blueprint for creating objects. An object is an instance of a class.

```
public class Car {
    // Fields
    String color;
    int year;

    // Method
    void start() {
        System.out.println("Car is starting");
    }
}
```

2. Main Method
- The entry point for any Java application is the 'main' method.

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.color = "Red";
        myCar.year = 2020;
        myCar.start();
    }
}
```

3. Data Types and Variables
- Java has various data types, including primitive types (e.g., 'int', 'char') and non-primitive types (e.g., 'String', arrays).

```
int number = 10;
double price = 99.99;
char grade = 'A';
boolean isJavaFun = true;
```

4. Control Flow Statements
- Conditional Statements: 'if', 'else if', 'else', 'switch'.
- Loops: 'for', 'while', 'do-while'.

5. Methods
- Methods def ne the behavior of objects. They can be called to perform operations or calculations.

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

Getting Started

1. Set Up Your Environment
- Install the JDK and set up your development environment. Popular IDEs include IntelliJ IDEA, Eclipse, and NetBeans.

2. Write Your First Program
- Create a simple "Hello, World!" program to get familiar with the syntax and basic structure of Java.

3. Explore Java Libraries
- Start using Java's built-in libraries to perform common tasks and build more complex applications.

We will see more about these topics in detail.

2. Basics of Java Programming

Basic Structure of a Java Program
A basic Java program consists of one or more classes. Each class can contain methods. The main method is the entry point of the application.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

To run a basic "Hello, World!" program in Java, you need to follow a few steps to compile and execute the code. Here's a step-by-step guide:

1. Write the Java Program
Create a f le named '**HelloWorld.java**' and include the following code:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
```

```
    }
}
```

2. Save the File
Make sure to save the f le with the '**.java**' extension. The f lename should match the name of the public class in the code ('**HelloWorld**').

3. Compile the Program
You need to compile the Java code to convert it into bytecode that the JVM can execute. Open your terminal or command prompt and navigate to the directory where 'HelloWorld.java' is located. Use the 'javac' command to compile the program:

```
javac HelloWorld.java
```

This command will generate a f le named **'HelloWorld.class**' in the same directory, which contains the bytecode.

4. Run the Program
To run the compiled Java program, use the 'java' command followed by the name of the class (without the '.class' extension):

```
java HelloWorld
```

You should see the output:

```
Hello, World!
```

5. Directives and Considerations
- Ensure Java is Installed: Make sure you have the Java Development Kit (JDK) installed on your system. You can check this by running 'java -version' and 'javac -version' in your terminal.

- Set Environment Variables: Ensure that the 'JAVA_HOME' environment variable is set correctly and that the 'bin' directory of the JDK is included in your system's 'PATH' variable. This allows you to run 'javac' and 'java' commands from any directory.

- Check File Name and Class Name: The f le name must match the class name exactly (case-sensitive). If your class is named 'HelloWorld', the f le must be named 'HelloWorld.java'.

Example Commands
Here's a summary of the commands you would use:

```
>> Navigate to the directory containing HelloWorld.java
cd path/to/your/directory

>> Compile the Java program
javac HelloWorld.java

>> Run the compiled Java program
java HelloWorld
```

By following these steps and ensuring your environment is correctly set up, you should be able to compile and run your Java program successfully.

3. Data Types in Java

Java has a rich set of data types, categorized into two main types: primitive data types and reference data types. Understanding these data types is fundamental for working ef ectively with Java.

1. Primitive Data Types

Primitive data types are the most basic data types in Java and are not objects. They are predef ned by the Java language and have a f xed size.

1.1. Integer Types

```
- 'byte': 8-bit signed integer.
  byte b = 127; // Range: -128 to 127

- 'short': 16-bit signed integer.
  short s = 32767; // Range: -32,768 to 32,767

- 'int': 32-bit signed integer.
  int i = 2_000_000_000; // Range: -2^31 to 2^31-1

- 'long': 64-bit signed integer.
  long l = 9_223_372_036_854_775_807L; // Range: -2^63 to 2^63-1
```

1.2. Floating-Point Types

```
- 'f oat': Single-precision 32-bit f oating-point.
f oat f = 3.14f; // Range: approximately ±3.40282347E+38F (6-7 decimal digits)

- 'double': Double-precision 64-bit f oating-point.
double d = 3.141592653589793; // Range: approximately ±1.79769313486231570E+308 (15
decimal digits)
```

1.3. Character Type

```
- 'char': 16-bit Unicode character.
char c = 'A'; // Range: '\u0000' (0) to '\uf f ' (65,535)
```

1.4. Boolean Type

```
- 'boolean': Represents one of two values: `true` or `false`.
  boolean isJavaFun = true;
```

2. Reference Data Types

Reference data types are used to refer to objects and arrays. They are not predef ned and can be created by def ning classes, interfaces, or arrays.

2.1. String

- 'String': Represents a sequence of characters. Unlike primitive types, `String` is a class in Java.

```
String greeting = "Hello, World!";
```

2.2. Arrays

- Arrays: Containers that hold a f xed number of values of a single type.

```
int[] numbers = {1, 2, 3, 4, 5}; // Array of integers
String[] names = {"Alice", "Bob", "Charlie"}; // Array of strings
```

2.3. User-Def ned Classes

- Classes: Def ne new data types by creating custom objects with f elds and methods.

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

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

```
// Creating an object
Person person = new Person("John", 30);
```

2.4. Interfaces

- Interfaces: Def ne methods that can be implemented by classes. Interfaces are used to def ne a contract for what a class can do.

```
public interface Drawable {
    void draw();
}

public class Circle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Summary

- Primitive Data Types: Include `byte`, `short`, `int`, `long`, `f oat`, `double`, `char`, and `boolean`. They are simple and have a f xed size.
- Reference Data Types: Include `String`, arrays, classes, and interfaces. They can represent more complex data structures and are managed via references.

Understanding and using these data types correctly is crucial for ef ective Java programming and for managing data ef iciently in your applications.

4. Variables and Constants

In Java, variables and constants are fundamental concepts used to store and manage data. Here's a detailed explanation of each:

1. Variables

Variables in Java are used to store data that can be modif ed during the execution of a program. Each variable has a specif c type, which determines what kind of data it can hold.

1.1. Variable Declaration and Initialization
-Declaration: Specif es the type and name of the variable.
-Initialization: Assigns a value to the variable.

```
int age; // Declaration
age = 25; // Initialization
```

You can also declare and initialize a variable in one line:

```
int age = 25;
```

1.2. Variable Types
-Local Variables: Declared inside a method or a block of code. They must be initialized before use.

```
public void displayAge() {
    int age = 30; // Local variable
    System.out.println(age);
}
```

-Instance Variables: Declared inside a class but outside any method. Each instance of the class has its own copy.

```
public class Person {
    String name; // Instance variable
}
```

-Class Variables (Static Variables): Declared with the 'static' keyword. There is only one copy of a static variable, shared among all instances of the class.

```
public class Counter {
    static int count = 0; // Static variable
}
```

1.3. Variable Naming Rules

- Names must start with a letter (a-z, A-Z), underscore (_), or dollar sign ($).
- Subsequent characters can be letters, digits (0-9), underscores, or dollar signs.
- Names cannot be reserved keywords (e.g., 'int', 'class', 'public').
- Naming conventions suggest using camelCase for variable names (e.g., 'myVariable').

2. Constants

Constants are variables whose values cannot be changed once they are assigned. In Java, constants are declared using the 'f nal' keyword.

2.1. Declaring Constants
Constants are typically declared as 'public static f nal' to make them accessible without needing an instance of the class and to ensure they cannot be modif ed.

```
public class MathConstants {
   public static f nal double PI = 3.14159; // Constant
}
```

2.2. Naming Conventions for Constants
- Constants are usually written in uppercase letters with underscores separating words.
- Example: 'MAX_VALUE', 'DEFAULT_TIMEOUT'.

2.3. Usage
You can use constants anywhere in your code after they have been def ned. Because they are 'static', you can access them directly through the class they are def ned in, without creating an instance.

```
public class Example {
   public static f nal int MAX_RETRIES = 5;

   public static void main(String[] args) {
      System.out.println("Max retries: " + MAX_RETRIES);
   }
}
```

Example program

```
public class RectangleArea {

   // Constant for the number of decimal places for the area calculation
   public static f nal int DECIMALS = 2;

   public static void main(String[] args) {
      // Variables for the dimensions of the rectangle
      double length = 10.5; // Length of the rectangle
      double width = 5.0;  // Width of the rectangle

      // Calculate the area of the rectangle
      double area = length * width;

      // Print the results
      System.out.println("Length: " + length);
      System.out.println("Width: " + width);
      System.out.println("Area: " + String.format("%." + DECIMALS + "f", area));
   }
```

```
}
```

Summary
-Variables: Used to store data that can change during program execution. They are categorized into local, instance, and static variables.
-Constants: Used to store data that should not change once initialized. They are declared with the 'f nal' keyword and are typically written in uppercase letters.

5. Operators

Operators in Java are special symbols or keywords that perform operations on variables and values. Java has several types of operators, each serving dif erent purposes. Here's a comprehensive overview:

1. Arithmetic Operators
Arithmetic operators are used to perform basic mathematical operations:

- Addition ('+'): Adds two values.

```
int sum = 10 + 5; // sum is 15
```

- Subtraction ('-'): Subtracts one value from another.

```
int dif erence = 10 - 5; // dif erence is 5
```

- Multiplication ('*'): Multiplies two values.

```
int product = 10 * 5; // product is 50
```

- Division ('/'): Divides one value by another.

```
int quotient = 10 / 5; // quotient is 2
```

- Modulus ('%'): Returns the remainder of division.

```
int remainder = 10 % 3; // remainder is 1
```

2. Relational Operators

Relational operators are used to compare two values:

- Equal to ('=='): Checks if two values are equal.

```
boolean isEqual = (10 == 5); // isEqual is false
```

- Not equal to ('!='): Checks if two values are not equal.

```
boolean isNotEqual = (10 != 5); // isNotEqual is true
```

- Greater than ('>'): Checks if one value is greater than another.

```
boolean isGreater = (10 > 5); // isGreater is true
```

- Less than ('<'): Checks if one value is less than another.

```
boolean isLess = (10 < 5); // isLess is false
```

- Greater than or equal to ('>='): Checks if one value is greater than or equal to another.

```
boolean isGreaterOrEqual = (10 >= 10); // isGreaterOrEqual is true
```

- Less than or equal to ('<='): Checks if one value is less than or equal to another.

```
boolean isLessOrEqual = (10 <= 5); // isLessOrEqual is false
```

3. Logical Operators

Logical operators are used to perform logical operations on boolean values:

- Logical AND ('&&'): Returns true if both operands are true.

```
boolean result = (10 > 5) && (5 < 3); // result is false
```

- Logical OR ('||'): Returns true if at least one of the operands is true.

```
boolean result = (10 > 5) || (5 < 3); // result is true
```

- Logical NOT ('!'): Inverts the boolean value.

```
boolean result = !(10 > 5); // result is false
```

4. Assignment Operators

Assignment operators are used to assign values to variables:

- Simple Assignment ('='): Assigns the value on the right to the variable on the left.

```
int a = 10;
```

- Add and Assign ('+='): Adds the right operand to the left operand and assigns the result to the left operand.

```
a += 5; // a is now 15
```

- Subtract and Assign ('-='): Subtracts the right operand from the left operand and assigns the result to the left operand.

```
a -= 3; // a is now 12
```

- Multiply and Assign ('*='): Multiplies the left operand by the right operand and assigns the result to the left operand.

```
a *= 2; // a is now 24
```

- Divide and Assign ('/='): Divides the left operand by the right operand and assigns the result to the left operand.

```
a /= 4; // a is now 6
```

- Modulus and Assign ('%='): Takes the modulus of the left operand by the right operand and assigns the result to the left operand.

```
a %= 4; // a is now 2
```

5. Unary Operators

Unary operators operate on a single operand:

- Unary Plus ('+'): Indicates a positive value (optional, often redundant).

```
int a = +10;
```

- Unary Minus ('-'): Negates the value.

```
int b = -10;
```

- Increment ('++'): Increases the value by 1.

```
int c = 5;
c++; // c is now 6
```

- Decrement ('--'): Decreases the value by 1.

```
int d = 5;
d--; // d is now 4
```

- Logical NOT ('!'): Inverts the boolean value.

```
boolean e = true;
e = !e; // e is now false
```

6. Bitwise Operators

Bitwise operators perform operations on the binary representation of integers:

- AND ('&'): Performs a bitwise AND operation.

```
int x = 5 & 3; // x is 1
```

- OR ('|'): Performs a bitwise OR operation.

```
int x = 5 | 3; // x is 7
```

- XOR ('^'): Performs a bitwise XOR operation.

```
int x = 5 ^ 3; // x is 6
```

- Complement ('~'): Inverts the bits.

```
int x = ~5; // x is -6
```

- Left Shift ('<<'): Shifts bits to the left, f lling with zeros.

```
int x = 5 << 1; // x is 10
```

- Right Shift ('>>'): Shifts bits to the right, preserving the sign bit.

```
int x = 5 >> 1; // x is 2
```

- Unsigned Right Shift ('>>>'): Shifts bits to the right, f lling with zeros regardless of the sign bit.

```
int x = -5 >>> 1; // x is 2147483642
```

7. Conditional (Ternary) Operator

The ternary operator is a shorthand for the 'if-else' statement:

- Ternary ('? :'): Evaluates a condition and returns one of two values based on the result.

```
int max = (a > b) ? a : b; // max is the greater of a and b
```

Example program using operators.

```
public class OperatorsDemo {

    public static void main(String[] args) {
        // Arithmetic Operators
        int a = 10;
        int b = 5;

        int sum = a + b; // Addition
        int dif erence = a - b; // Subtraction
        int product = a * b; // Multiplication
        int quotient = a / b; // Division
        int remainder = a % b; // Modulus

        System.out.println("Arithmetic Operators:");
        System.out.println("a + b = " + sum);
        System.out.println("a - b = " + dif erence);
        System.out.println("a * b = " + product);
        System.out.println("a / b = " + quotient);
        System.out.println("a % b = " + remainder);

        // Relational Operators
        boolean isEqual = (a == b); // Equal to
        boolean isNotEqual = (a != b); // Not equal to
        boolean isGreater = (a > b); // Greater than
        boolean isLess = (a < b); // Less than
        boolean isGreaterOrEqual = (a >= b); // Greater than or equal to
        boolean isLessOrEqual = (a <= b); // Less than or equal to

        System.out.println("\nRelational Operators:");
        System.out.println("a == b: " + isEqual);
        System.out.println("a != b: " + isNotEqual);
        System.out.println("a > b: " + isGreater);
        System.out.println("a < b: " + isLess);
        System.out.println("a >= b: " + isGreaterOrEqual);
        System.out.println("a <= b: " + isLessOrEqual);

        // Logical Operators
        boolean condition1 = (a > 0);
        boolean condition2 = (b < 10);

        boolean andResult = condition1 && condition2; // Logical AND
        boolean orResult = condition1 || condition2; // Logical OR
        boolean notResult = !condition1; // Logical NOT

        System.out.println("\nLogical Operators:");
        System.out.println("condition1 && condition2: " + andResult);
        System.out.println("condition1 || condition2: " + orResult);
```

```
        System.out.println("!condition1: " + notResult);

        // Assignment Operators
        int x = 10;

        x += 5; // Add and assign
        System.out.println("\nAssignment Operators:");
        System.out.println("x += 5: " + x);

        x -= 3; // Subtract and assign
        System.out.println("x -= 3: " + x);

        x *= 2; // Multiply and assign
        System.out.println("x *= 2: " + x);

        x /= 4; // Divide and assign
        System.out.println("x /= 4: " + x);

        x %= 3; // Modulus and assign
        System.out.println("x %= 3: " + x);
    }
}
```

Summary
- Arithmetic Operators: Perform mathematical operations ('+', '-', '*', '/', '%').
- Relational Operators: Compare values ('==', '!=', '>', '<', '>=', '<=').
- Logical Operators: Perform logical operations ('&&', '||', '!').
- Assignment Operators: Assign values and perform operations ('=', '+=', '-=', '*=', '/=', '%=').
- Unary Operators: Operate on a single operand ('+', '-', '++', '--', '!').
- Bitwise Operators: Perform bit-level operations ('&', '|', '^', '~', '<<', '>>', '>>>').
- Conditional (Ternary) Operator: Shortened form of 'if-else' ('? :').

6. Control Structures

Control structures in Java are used to control the f ow of execution in a program based on certain conditions or to repeat certain operations. Here's a detailed overview of the main control structures:

1.Decision-Making Statements

Decision-making statements are used to execute dif erent blocks of code based on certain conditions.

1.1. 'if' Statement
The 'if' statement executes a block of code if a specif ed condition is true.

```
if (condition) {
    // Code to be executed if the condition is true
}
```

Example:

```
int age = 20;
if (age >= 18) {
   System.out.println("You are an adult.");
}
```

1.2. 'if-else' Statement

The 'if-else' statement executes one block of code if the condition is true, and another block if the condition is false.

```
if (condition) {
   // Code to be executed if the condition is true
} else {
   // Code to be executed if the condition is false
}
```

Example:

```
int age = 16;
if (age >= 18) {
   System.out.println("You are an adult.");
} else {
   System.out.println("You are a minor.");
}
```

1.3. 'if-else if-else' Statement

The 'if-else if-else' statement allows multiple conditions to be checked sequentially.

```
if (condition1) {
   // Code to be executed if condition1 is true
} else if (condition2) {
   // Code to be executed if condition2 is true
} else {
   // Code to be executed if none of the conditions are true
}
```

Example:

```
int score = 85;
if (score >= 90) {
   System.out.println("Grade: A");
} else if (score >= 80) {
   System.out.println("Grade: B");
} else if (score >= 70) {
   System.out.println("Grade: C");
} else {
   System.out.println("Grade: D");
}
```

1.4. 'switch' Statement

The 'switch' statement executes one block of code based on the value of an expression.

```
switch (expression) {
   case value1:
      // Code to be executed if expression equals value1
      break;
```

```
  case value2:
    // Code to be executed if expression equals value2
    break;
  // You can have any number of case statements
  default:
    // Code to be executed if no case matches
}
```

Example:

```
int day = 3;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  default:
    System.out.println("Invalid day");
}
```

2.Looping Statements

Looping statements are used to repeat a block of code multiple times.

2.1. 'for' Loop

The 'for' loop is used when the number of iterations is known beforehand.

```
for (initialization; condition; update) {
  // Code to be executed in each iteration
}
```

Example:

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

2.2. 'while' Loop

The 'while' loop is used when the number of iterations is not known and depends on a condition.

```
while (condition) {
  // Code to be executed as long as condition is true
}
```

Example:

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
```

```
}
```

2.3. 'do-while' Loop

The 'do-while' loop is similar to the 'while' loop but guarantees that the code block will be executed at least once.

```
do {
   // Code to be executed
} while (condition);
```

Example:

```
int i = 0;
do {
   System.out.println(i);
   i++;
} while (i < 5);
```

3.Control Flow Modif ers

Control f ow modif ers alter the f ow of control in loops and switch statements.

3.1. 'break' Statement

The 'break' statement exits from the loop or switch statement immediately.

```
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break; // Exits the loop when i is 5
   }
   System.out.println(i);
}
```

3.2. 'continue' Statement

The 'continue' statement skips the current iteration of the loop and proceeds to the next iteration.

```
for (int i = 0; i < 10; i++) {
   if (i % 2 == 0) {
      continue; // Skips the iteration when i is even
   }
   System.out.println(i);
}
```

Example programs:

Using Decision making statements

```
public class DecisionMakingDemo {

   public static void main(String[] args) {
      int score = 85; // Student's score

      // If-else statement to determine grade
      if (score >= 90) {
         System.out.println("Grade: A");
      } else if (score >= 80) {
         System.out.println("Grade: B");
```

```
    } else if (score >= 70) {
      System.out.println("Grade: C");
    } else if (score >= 60) {
      System.out.println("Grade: D");
    } else {
      System.out.println("Grade: F");
    }

    // Switch statement to determine feedback based on grade
    char grade = getGrade(score); // Get the grade based on the score

    System.out.println("\nFeedback based on grade:");
    switch (grade) {
      case 'A':
        System.out.println("Excellent work!");
        break;
      case 'B':
        System.out.println("Good job!");
        break;
      case 'C':
        System.out.println("Fair performance.");
        break;
      case 'D':
        System.out.println("Needs improvement.");
        break;
      case 'F':
        System.out.println("Failed. Better luck next time.");
        break;
      default:
        System.out.println("Invalid grade.");
        break;
    }
  }

  // Method to determine the grade based on the score
  public static char getGrade(int score) {
    if (score >= 90) {
      return 'A';
    } else if (score >= 80) {
      return 'B';
    } else if (score >= 70) {
      return 'C';
    } else if (score >= 60) {
      return 'D';
    } else {
      return 'F';
    }
  }
}
```

Example using Looping Statements

```
public class LoopingDemo {

   public static void main(String[] args) {
      // Using a for loop
      System.out.println("Using for loop:");
      for (int i = 1; i <= 5; i++) {
         System.out.println(i);
      }

      // Using a while loop
      System.out.println("\nUsing while loop:");
      int j = 1;
      while (j <= 5) {
         System.out.println(j);
         j++;
      }

      // Using a do-while loop
      System.out.println("\nUsing do-while loop:");
      int k = 1;
      do {
         System.out.println(k);
         k++;
      } while (k <= 5);
   }
}
```

Example program using Control Flow Modif ers

```
public class ControlFlowModif ersDemo {
   public static void main(String[] args) {
      System.out.println("Using break in nested loops:");

      // Using break to exit from the inner loop
      outerLoop: // Label for outer loop
      for (int i = 1; i <= 3; i++) {
         for (int j = 1; j <= 3; j++) {
            if (j == 2) {
               break outerLoop; // Exits from the outer loop
            }
            System.out.println("i = " + i + ", j = " + j);
         }
      }

      System.out.println("\nUsing continue in nested loops:");

      // Using continue to skip the current iteration of the inner loop
      for (int i = 1; i <= 3; i++) {
         for (int j = 1; j <= 3; j++) {
            if (j == 2) {
               continue; // Skips the rest of the inner loop and continues with the next iteration of the
outer loop
```

```
            }
            System.out.println("i = " + i + ", j = " + j);
        }
    }
  }
}
```

Summary
-Decision-Making Statements: 'if', 'if-else', 'if-else if-else', 'switch'.
-Looping Statements: 'for', 'while', 'do-while'.
-Control Flow Modif ers: 'break', 'continue'.

7. Arrays and Strings

Arrays and strings are fundamental data structures in Java. Here's a beginner-friendly overview of each:

Arrays
What is an Array?
An array is a container that holds a f xed number of values of a single type. It allows you to store multiple values in a single variable, instead of declaring separate variables for each value.

Declaring and Initializing Arrays
1. Declaring an Array:

```
int[] numbers; // Declaration of an array of integers
```

2.Initializing an Array:
-Static Initialization

```
int[] numbers = {1, 2, 3, 4, 5}; // Initialization with values
```

-Dynamic Initialization:

```
int[] numbers = new int[5]; // Creates an array of size 5 with default values (0 for int)
```

3.Accessing Array Elements:

```
numbers[0] = 1; // Assigns 1 to the f rst element
int f rstNumber = numbers[0]; // Retrieves the f rst element
```

Example Program:

```
public class ArrayDemo {
    public static void main(String[] args) {
        // Declare and initialize an array
        int[] numbers = {10, 20, 30, 40, 50};

        // Access and print array elements
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }
    }
}
```

Strings

What is a String?

A string is a sequence of characters. In Java, strings are objects of the 'String' class and are used to represent text.

Creating Strings

1.Using String Literals:

```
String greeting = "Hello, World!";
```

2.Using the 'new' Keyword:

```
String greeting = new String("Hello, World!");
```

Common String Methods

1.Length of a String:

```
int length = greeting.length(); // Gets the length of the string
```

2.Concatenation:

```
String fullName = "John" + " " + "Doe"; // Concatenates strings
```

3.Substring:

```
String substring = greeting.substring(7, 12); // Extracts substring from index 7 to 11
```

4.Uppercase and Lowercase:

```
String upperCase = greeting.toUpperCase(); // Converts string to uppercase
String lowerCase = greeting.toLowerCase(); // Converts string to lowercase
```

5.Comparison:

```
boolean isEqual = greeting.equals("Hello, World!"); // Checks if strings are equal
```

Example Program:

```
public class StringDemo {
    public static void main(String[] args) {
        // Create and manipulate strings
        String greeting = "Hello, World!";

        // Print length of the string
        System.out.println("Length of the string: " + greeting.length());

        // Convert to uppercase and print
        System.out.println("Uppercase: " + greeting.toUpperCase());

        // Extract and print a substring
        String substring = greeting.substring(7, 12);
        System.out.println("Substring: " + substring);
    }
}
```

Summary

-Arrays:
 -Purpose: Store multiple values of the same type.
 -Syntax: 'dataType[] arrayName = new dataType[size];' or 'dataType[] arrayName = {values};'
 -Access: Use an index ('arrayName[index]').

-Strings:
 -Purpose: Represent and manipulate text.
 -Creation: 'String variableName = "text";' or 'new String("text");'
 -Methods: 'length()', 'toUpperCase()', 'substring()', 'equals()', etc.

8. Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a fundamental concept in Java and many other programming languages. It helps organize code in a way that's modular, reusable, and easy to understand. Imagine you're building a house. Instead of thinking about each brick individually, you think about rooms. Each room has its own purpose (bedroom for sleeping, kitchen for cooking). This is essentially what OOP does in programming.

Core Principles of OOP

1. Classes and Objects
- Class: A class is a blueprint or template for creating objects. It def nes a type of object, including its attributes (data) and methods (functions or behaviors).
- Object: An object is an instance of a class. It's a specif c realization of a class with actual values for its attributes. These are the building blocks of OOP. They represent real-world entities like a car, a person, or a bank account. Each object has its own characteristics (data) and actions (methods)

```
// Def ne a class named 'Car'
public class Car {
    // Attributes (f elds) of the class
    String color;
    String model;

    // Method of the class
    void startEngine() {
        System.out.println("The engine is starting.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the 'Car' class
        Car myCar = new Car();

        // Set the attributes
        myCar.color = "Red";
        myCar.model = "Toyota";

        // Call the method
```

```
        myCar.startEngine();
    }
}
```

2. Encapsulation

- Encapsulation is the concept of wrapping data (attributes) and methods (functions) into a single unit, or class. It hides the internal state of an object from the outside world and only exposes a controlled interface.
- Bundling data (attributes) and methods (functions) that operate on the data within a single unit (object).
- Protects data from accidental modif cation.
- Example: A car has properties like color, model, and methods like start, stop, accelerate.

```
public class Person {
    // Private attributes (f elds)
    private String name;
    private int age;

    // Public method to set the name
    public void setName(String name) {
        this.name = name;
    }

    // Public method to get the name
    public String getName() {
        return name;
    }

    // Public method to set the age
    public void setAge(int age) {
        this.age = age;
    }

    // Public method to get the age
    public int getAge() {
        return age;
    }
}
public class Main {
    public static void main(String[] args) {
        // Create an object of the 'Person' class
        Person person = new Person();

        // Set attributes using public methods
        person.setName("Alice");
        person.setAge(30);

        // Get and print attributes using public methods
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
```

```
}
```

3. Inheritance

- Inheritance allows one class (the child or subclass) to inherit the attributes and methods of another class (the parent or superclass). This promotes code reuse and establishes a hierarchy between classes.
- Creating new classes (child classes) from existing ones (parent classes).
- Child classes inherit properties and methods from the parent class.
- Example: A Dog class inherits properties and methods from an Animal class.

```
// Base class
public class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
// Subclass inheriting from Animal
public class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        // Create an object of the subclass
        Dog myDog = new Dog();

        // Call methods from both the base class and the subclass
        myDog.eat();
        myDog.bark();
    }
}
```

4. Polymorphism

- Polymorphism means "many forms" and allows objects to be treated as instances of their parent class rather than their actual class. It enables one method to do dif erent things based on the object it is acting upon. Example: Dif erent animals can make dif erent sounds, but they all share the makeSound() method.

```
  // Base class
public class Animal {
    void makeSound() {
        System.out.println("This animal makes a sound.");
    }
}

// Subclass overriding the method
public class Cat extends Animal {
    @Override
    void makeSound() {
```

```
        System.out.println("The cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create objects of Animal and Cat
        Animal myAnimal = new Animal();
        Animal myCat = new Cat();

        // Call the method
        myAnimal.makeSound(); // Outputs: This animal makes a sound.
        myCat.makeSound();    // Outputs: The cat meows.
    }
}
```

5. Abstraction

Abstraction is a fundamental concept in OOP that allows you to hide complex implementation details and show only the necessary features of an object. It helps in focusing on what an object does rather than how it does it.

1. Abstract Classes:

- An abstract class is a class that cannot be instantiated on its own and may contain abstract methods (methods without implementation) that must be implemented by subclasses.

```
// Abstract class
abstract class Animal {
    // Abstract method (does not have a body)
    abstract void makeSound();

    // Regular method
    void sleep() {
        System.out.println("This animal is sleeping.");
    }
}

// Subclass that inherits from Animal
class Dog extends Animal {
    // Providing implementation of the abstract method
    @Override
    void makeSound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the subclass
        Dog myDog = new Dog();

        // Call methods
```

```
        myDog.makeSound(); // Outputs: The dog barks.
        myDog.sleep();     // Outputs: This animal is sleeping.
    }
}
```

2. Interfaces:

- An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance f elds or constructors. Classes that implement an interface must provide implementations for all of its methods.

```
// Interface
interface Animal {
    // Abstract method (does not have a body)
    void makeSound();

    // Default method
    default void sleep() {
        System.out.println("This animal is sleeping.");
    }
}

// Class that implements the interface
class Cat implements Animal {
    // Providing implementation of the abstract method
    @Override
    public void makeSound() {
        System.out.println("The cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of the implementing class
        Cat myCat = new Cat();

        // Call methods
        myCat.makeSound(); // Outputs: The cat meows.
        myCat.sleep();     // Outputs: This animal is sleeping.
    }
}
```

Summary

1. Classes and Objects: Classes def ne the structure and behavior of objects. Objects are instances of classes.
2. Encapsulation: Encapsulates data and methods within a class, providing a controlled interface.
3. Inheritance: Allows a class to inherit attributes and methods from another class, enabling code reuse.
4. Polymorphism: Allows methods to perform dif erent tasks based on the object it is working with, supporting method overriding and method overloading.

5. Abstract Classes: Def ne a common base with some methods implemented and some left to be implemented by subclasses. They allow you to def ne a template for other classes.
6. Interfaces: Def ne a contract that implementing classes must follow. They allow dif erent classes to implement the same set of methods, regardless of their position in the class hierarchy.

9. Method Overloading

Method Overloading in Java is a feature that allows a class to have more than one method with the same name, but with dif erent parameters. It is a form of compile-time (or static) polymorphism, which enables methods to perform similar functions with dif erent types or numbers of arguments.

Key Points of Method Overloading:

1. Same Method Name:
 - Overloaded methods have the same name but must dif er in the type, number, or both of their parameters.

2. Dif erent Parameters:
 - The method signature (method name and parameter list) must be dif erent for each overloaded method. This means the parameter list must vary in number, type, or both.

3. Return Type:
 - The return type alone is not enough to dif erentiate overloaded methods. They must dif er in their parameter lists. However, you can have dif erent return types for overloaded methods as long as their parameter lists are dif erent.

4. Compile-Time Polymorphism:
 - Method overloading is resolved at compile time. The compiler determines which method to call based on the method signature and arguments used.

Example 1: Dif erent Number of Parameters

```
public class OverloadExample {

  // Method with one parameter
  public void display(int a) {
    System.out.println("Display method with one parameter: " + a);
  }

  // Overloaded method with two parameters
  public void display(int a, int b) {
    System.out.println("Display method with two parameters: " + a + ", " + b);
  }

  public static void main(String[] args) {
    OverloadExample obj = new OverloadExample();
    obj.display(10);      // Calls the method with one parameter
    obj.display(10, 20);  // Calls the method with two parameters
  }
```

```
}
```

Example 2: Dif erent Types of Parameters

```
public class OverloadExample {

    // Method with int parameter
    public void show(int a) {
        System.out.println("Show method with int: " + a);
    }

    // Overloaded method with double parameter
    public void show(double a) {
        System.out.println("Show method with double: " + a);
    }

    // Overloaded method with string parameter
    public void show(String a) {
        System.out.println("Show method with string: " + a);
    }

    public static void main(String[] args) {
        OverloadExample obj = new OverloadExample();
        obj.show(10);       // Calls the method with int parameter
        obj.show(10.5);     // Calls the method with double parameter
        obj.show("Hello");  // Calls the method with string parameter
    }
}
```

Example 3: Dif erent Parameter Order

```
public class OverloadExample {

    // Method with int and double parameters
    public void print(int a, double b) {
        System.out.println("Print method with int and double: " + a + ", " + b);
    }

    // Overloaded method with double and int parameters
    public void print(double a, int b) {
        System.out.println("Print method with double and int: " + a + ", " + b);
    }

    public static void main(String[] args) {
        OverloadExample obj = new OverloadExample();
        obj.print(10, 10.5);  // Calls the method with int and double parameters
        obj.print(10.5, 10);  // Calls the method with double and int parameters
    }
}
```

Summary

Method Overloading allows a class to have multiple methods with the same name but dif erent parameters.
- Overloaded methods must dif er in the number or type of parameters.
- Return type alone cannot be used to overload methods; the method signature must be dif erent.
- Method overloading is resolved at compile time and enhances code readability and reusability.

10. Method Overriding

Method Overriding is a fundamental concept in Java that is used to achieve runtime polymorphism. It allows a subclass to provide a specif c implementation for a method that is already def ned in its superclass. When a method is overridden, the subclass's version of the method is called, even if the method is invoked on a superclass reference.

Key Points of Method Overriding

1. Same Method Signature:
  - The method in the subclass must have the same name, return type, and parameter list as the method in the superclass. This is necessary to override the method.

2. Access Modif er:
  - The overriding method in the subclass cannot have a more restrictive access modif er than the method in the superclass. For example, if the superclass method is 'public', the overriding method in the subclass must also be 'public'. It can be 'protected' or package-private, but not 'private'.

3. Return Type:
  - The return type of the overriding method must be the same as, or a subtype of, the return type of the method in the superclass. This is known as covariant return type.

4. '@Override' Annotation:
  - The '@Override' annotation is used to indicate that a method is intended to override a method in the superclass. It helps the compiler and improves code readability by making it clear that the method is overriding a superclass method.

5. Super Keyword:
  - Inside the overriding method, you can use the 'super' keyword to call the superclass's version of the method if needed.

6. Exception Handling:
  - The overriding method cannot throw more checked exceptions than the method in the superclass. It can throw fewer exceptions or no exceptions at all.

Here's a simple example to illustrate method overriding:

```
// Superclass
class Animal {
  // Method to be overridden
  void makeSound() {
    System.out.println("Animal makes a sound");
  }
```

```
}

// Subclass
class Dog extends Animal {
   // Overriding the method from Animal class
   @Override
   void makeSound() {
      System.out.println("Dog barks");
   }
}

public class Main {
   public static void main(String[] args) {
      // Create an instance of Dog
      Animal myDog = new Dog();

      // Call the overridden method
      myDog.makeSound();  // Outputs: Dog barks
   }
}
```

- Superclass: 'Animal' has a method 'makeSound()'.
- Subclass: 'Dog' extends 'Animal' and overrides the 'makeSound()' method.
- Method Call: When 'makeSound()' is called on an 'Animal' reference that actually points to a 'Dog' object, the 'Dog''s version of 'makeSound()' is executed.

Important Considerations

- Polymorphism: Method overriding is key to achieving runtime polymorphism in Java, where the method that gets executed depends on the actual object type, not the reference type.
- Overriding vs. Overloading: Method overriding involves redef ning a method in a subclass, while method overloading involves def ning multiple methods with the same name but dif erent parameter lists within the same class.
- Super Method Call: The 'super' keyword allows calling the superclass version of the overridden method if needed.

Summary

- Method Overriding allows a subclass to provide a specif c implementation for a method already def ned in its superclass.
- It must have the same name, return type, and parameters as the method in the superclass.
- The '@Override' annotation helps indicate that a method is overriding a superclass method.
- Method overriding supports runtime polymorphism and dynamic method dispatch.

11. Using “Super” keyword

The 'super' keyword in Java is used in subclasses to refer to their superclass. It provides a way to access and interact with members (f elds, methods, and constructors) of the superclass from within the subclass. The 'super' keyword can be used in various contexts, and understanding its use is important for ef ective inheritance and polymorphism in Java.

Key Uses of 'super' Keyword

1. Accessing Superclass Methods
- You can use 'super' to call a method from the superclass that has been overridden in the subclass. This is useful when you want to invoke the original implementation of a method in addition to or instead of the subclass's version.

Example:

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

class Dog extends Animal {
    @Override
    void makeSound() {
        super.makeSound(); // Calls the superclass method
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();
    }
}
```

Output

```
Animal makes a sound
Dog barks
```

2. Accessing Superclass Fields
- You can use 'super' to refer to f elds in the superclass, especially when the subclass has a f eld with the same name. It helps to distinguish between the subclass's f eld and the superclass's f eld.

Example:

```
class Animal {
    String color = "White";
}

class Dog extends Animal {
    String color = "Brown";

    void printColor() {
        System.out.println("Dog color: " + color);       // Refers to the subclass f eld
        System.out.println("Animal color: " + super.color); // Refers to the superclass f eld
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.printColor();
    }
}
```

- Output:

```
Dog color: Brown
Animal color: White
```

3. Calling Superclass Constructor

- You can use 'super' to invoke the constructor of the superclass from the subclass constructor. This is important for initializing the superclass part of the object properly.

Example:

```
class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }
}

class Dog extends Animal {
    Dog() {
        super(); // Calls the superclass constructor
        System.out.println("Dog constructor");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
    }
}
```

- Output:

```
Animal constructor
Dog constructor
```

4. Accessing Superclass Method from Constructor

- In a subclass constructor, you can use 'super' to call methods from the superclass if you need to perform initialization or setup that depends on superclass methods.

Example:

```
class Animal {
    Animal() {
        System.out.println("Animal constructor");
        makeSound(); // Calls the subclass method
    }

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

```
}

class Dog extends Animal {
    Dog() {
        System.out.println("Dog constructor");
    }

    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
    }
}
```

- Output:

```
Animal constructor
Dog barks
Dog constructor
```

Summary

- 'super' Keyword: Refers to the superclass of the current object.
- Accessing Methods: Use 'super.methodName()' to call a method from the superclass.
- Accessing Fields: Use 'super.f eldName' to access a f eld in the superclass if it is shadowed by a subclass f eld.
- Calling Superclass Constructor: Use 'super()' to invoke a constructor of the superclass from a subclass constructor.
- Constructor Calls: The 'super' keyword helps ensure proper initialization by calling superclass constructors and methods.

12. Interfaces

In Java, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces are used to specify a set of methods that implementing classes must provide, promoting a form of abstraction and enabling multiple inheritance of type.

Key Characteristics of Interfaces

1. Abstract Methods:

- Interfaces primarily declare abstract methods—methods without a body. Any class that implements an interface must provide concrete implementations for these methods.

2. Default Methods:

- Introduced in Java 8, default methods in interfaces have a body. They provide a default implementation, so implementing classes do not need to override them unless they want to provide a specif c implementation.

3. Static Methods:
- Interfaces can also contain static methods with a body. These methods are not inherited by implementing classes but can be called on the interface itself.

4. Constants:
- Interfaces can declare constants (static f nal variables). All f elds in an interface are implicitly 'public', 'static', and 'f nal'.

5. No Constructor:
- Interfaces cannot have constructors, as they cannot be instantiated directly.

6. Multiple Inheritance:
- Java supports multiple inheritance through interfaces. A class can implement multiple interfaces, allowing it to inherit methods from more than one source.

7. Inheritance:
- Interfaces can extend other interfaces, allowing for the creation of more complex method sets. An interface can extend multiple interfaces.

Syntax

```
// Def ne an interface
interface Animal {
  // Abstract method (does not have a body)
  void eat();

  // Default method (with a body)
  default void sleep() {
    System.out.println("This animal sleeps");
  }

  // Static method (with a body)
  static void breath() {
    System.out.println("This animal breathes");
  }
}
```

Implementing an Interface
A class that implements an interface must provide implementations for all abstract methods declared in the interface.

```
class Dog implements Animal {
  // Implementing the abstract method
  @Override
  public void eat() {
    System.out.println("Dog eats");
  }

  // Optionally override the default method
  @Override
  public void sleep() {
    System.out.println("Dog sleeps");
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat();   // Outputs: Dog eats
        myDog.sleep(); // Outputs: Dog sleeps

        // Call the static method from the interface
        Animal.breath(); // Outputs: This animal breathes
    }
}
```

Key Uses of Interfaces

1. Abstraction:
  - Interfaces provide a way to def ne abstract types. They allow you to specify what methods a class must implement, without specifying how they are implemented.

2. Multiple Inheritance:
  - Interfaces allow a class to implement multiple interfaces, thereby supporting a form of multiple inheritance that Java does not support with classes.

3. Loose Coupling:
  - By programming to an interface rather than a specif c implementation, code becomes more f exible and easier to maintain.

4. Design Patterns:
  - Interfaces are fundamental in many design patterns, such as the Strategy, Observer, and Factory patterns.

5. API Design:
  - Interfaces can be used to def ne APIs, allowing dif erent implementations to be swapped out or extended without modifying the code that uses them.

Summary
- Interface: A reference type in Java that can contain abstract methods, default methods, static methods, and constants.
- Abstract Methods: Must be implemented by any class that implements the interface.
- Default Methods: Provide a default implementation that can be optionally overridden.
- Static Methods: Belong to the interface itself, not to any instance.
- Constants: Fields in interfaces are implicitly public, static, and f nal.
- Implementation: A class implements an interface by providing concrete implementations of all abstract methods.

13. Abstract Class vs. Interface

1. Purpose and Usage

- Abstract Class:

- Used to provide a common base for related classes.
- Can include both abstract methods (without implementation) and concrete methods (with implementation).
- Used when you want to share code among closely related classes and create a default behavior.

- Interface:

- Used to def ne a contract that other classes must adhere to.
- All methods in an interface are implicitly abstract (before Java 8) or can have default and static implementations (since Java 8).
- Used to def ne capabilities that can be shared across unrelated classes and to support multiple inheritance.

2. Method Implementation

- Abstract Class:
- Can have abstract methods (methods without implementation) and concrete methods (methods with implementation).
- Example:

```
abstract class Animal {
    abstract void makeSound(); // Abstract method
    void sleep() {          // Concrete method
        System.out.println("This animal is sleeping.");
    }
}
```

- Interface:
- All methods are abstract by default (before Java 8), but since Java 8, interfaces can have default methods (methods with implementation) and static methods.
- Example:

```
interface Animal {
    void makeSound();       // Abstract method

    default void sleep() {  // Default method
        System.out.println("This animal is sleeping.");
    }
}
```

3. Fields (Attributes)

- Abstract Class:
- Can have instance variables (f elds) with any access modif er ('private', 'protected', 'public').
- Fields can be initialized or uninitialized.
- Example:

```
abstract class Animal {
    String color; // Instance variable
    abstract void makeSound();
}
```

- Interface:
  - Can only have public, static, and f nal f elds (constants).
  - Fields in an interface are implicitly 'public', 'static', and 'f nal' (constants).
  - Example:

```
interface Animal {
    int LEGS = 4; // Constant (public, static, f nal)
    void makeSound();
}
```

4. Constructor

- Abstract Class:
  - Can have constructors to initialize its f elds.
  - Example:

```
abstract class Animal {
    String color;
    Animal(String color) {
        this.color = color;
    }
    abstract void makeSound();
}
```

- Interface:
  - Cannot have constructors because you cannot instantiate an interface directly.
  - Example:

```
interface Animal {
    void makeSound();
}
```

5. Inheritance and Implementation

- Abstract Class:
  - A class can inherit from only one abstract class (single inheritance).
  - Example:

```
class Dog extends Animal {
    void makeSound() {
        System.out.println("The dog barks.");
    }
}
```

- Interface:
  - A class can implement multiple interfaces (multiple inheritance).
  - Example:

```
interface Runnable {
    void run();
}

interface Flyable {
    void f y();
}
```

```
class Superhero implements Runnable, Flyable {
    public void run() {
        System.out.println("Running fast!");
    }

    public void f y() {
        System.out.println("Flying high!");
    }
}
```

6. Multiple Inheritance
- Abstract Class:
  - Java supports single inheritance of abstract classes.
  - A class can inherit from only one abstract class.

- Interface:
  - Java supports multiple inheritance of interfaces.
  - A class can implement multiple interfaces.

Summary
- Abstract Class:
  - Provides a base class with common functionality.
  - Can have both abstract and concrete methods.
  - Can have constructors and instance variables.
  - Supports single inheritance.

- Interface:
  - Def nes a contract for what a class should do.
  - Methods are abstract by default (or default/static since Java 8).
  - Can only have constants (public, static, f nal f elds).
  - Does not support constructors.
  - Supports multiple inheritance.

14. Constructors and Destructors

In Java, constructors and destructors are fundamental concepts related to object creation and destruction. Here's a detailed look at both:

Constructors
A constructor in Java is a special method that is automatically called when an object is instantiated. It is used to initialize the newly created object. Constructors have the same name as the class and do not have a return type.

Key Points about Constructors

1. Initialization: Constructors are used to set initial values for object attributes and perform any setup required when an object is created.

2. Syntax:

- Default Constructor: A constructor with no parameters. If no constructor is provided, Java provides a default constructor.
- Parameterized Constructor: A constructor that takes arguments to initialize object attributes.

3. Overloading: You can have multiple constructors in a class with dif erent parameter lists (constructor overloading).

4. Invocation: Constructors are called using the 'new' keyword when creating an object.
Example

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

  // Default constructor
  Person() {
    name = "Unknown";
    age = 0;
  }

  // Parameterized constructor
  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

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

public class Main {
  public static void main(String[] args) {
    Person person1 = new Person();           // Calls default constructor
    Person person2 = new Person("Alice", 30);    // Calls parameterized constructor

    person1.display();  // Outputs: Name: Unknown, Age: 0
    person2.display();  // Outputs: Name: Alice, Age: 30
  }
}
```

Destructors
Java does not have destructors in the traditional sense. In languages like C++, destructors are used to clean up resources before an object is destroyed. However, Java has an automatic garbage collection mechanism that manages memory and cleans up unused objects.

Resource Cleanup in Java

Instead of destructors, Java uses the following approaches for resource cleanup:

1. Finalize Method:

- Java provides a 'f nalize()' method in the 'Object' class, which can be overridden to perform cleanup operations before an object is garbage collected.
- However, relying on 'f nalize()' is generally discouraged due to its unpredictable nature and performance overhead. It's better to use other resource management techniques.

```
class Resource {
    @Override
    protected void f nalize() throws Throwable {
        try {
            // Cleanup code, like closing f le streams or database connections
            System.out.println("Resource is being cleaned up");
        } f nally {
            super.f nalize();
        }
    }
}
```

2. Try-With-Resources Statement:

- For managing resources like f les or database connections, Java provides the try-with-resources statement introduced in Java 7. It ensures that resources are closed automatically when the block is exited.
- Classes that need to be managed this way must implement the 'AutoCloseable' interface or its sub-interface 'java.io.Closeable'.

```
try (FileReader f leReader = new FileReader("f le.txt")) {
    // Use the f leReader
} catch (IOException e) {
    e.printStackTrace();
}
// f leReader is automatically closed here
```

3. Explicit Cleanup:

- For non-automatic resources, you can explicitly provide cleanup methods in your class and call them when the object is no longer needed.

```
class Resource {
    void cleanup() {
        // Perform cleanup operations
        System.out.println("Resource is cleaned up");
    }
}

public class Main {
    public static void main(String[] args) {
        Resource resource = new Resource();
        try {
            // Use the resource
        } f nally {
            resource.cleanup(); // Explicitly call cleanup method
        }
```

```
    }
  }
```

Summary
- Constructors:
  - Special methods called when an object is instantiated.
  - Used to initialize object attributes.
  - Can be default or parameterized, and can be overloaded.

- Destructors:
  - Java does not have destructors in the traditional sense.
  - Resource cleanup is typically handled by garbage collection, 'f nalize()' (discouraged), try-with-resources statement, or explicit cleanup methods.

15. Inheritance

Inheritance is a mechanism that allows one class (subclass or child class) to inherit the f elds and methods of another class (superclass or parent class). This promotes code reuse and establishes a hierarchical relationship between classes. Java supports several types of inheritance, though it has some restrictions and features that dif erentiate it from other languages.

Types of Inheritance in Java

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance (through Interfaces)
5. Hybrid Inheritance (not directly supported)

1. Single Inheritance
Single Inheritance occurs when a class (subclass) inherits from another class (superclass). This is the simplest form of inheritance.

Example

```
class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
```

```
    myDog.eat(); // Inherited method
    myDog.bark(); // Subclass method
  }
}
```

2. Multilevel Inheritance

Multilevel Inheritance occurs when a class is derived from another derived class. This creates a chain of inheritance.

Example

```
class Animal {
  void eat() {
    System.out.println("Animal eats");
  }
}

class Mammal extends Animal {
  void breathe() {
    System.out.println("Mammal breathes");
  }
}

class Dog extends Mammal {
  void bark() {
    System.out.println("Dog barks");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog myDog = new Dog();
    myDog.eat();   // From Animal class
    myDog.breathe(); // From Mammal class
    myDog.bark();  // Own method
  }
}
```

3. Hierarchical Inheritance

Hierarchical Inheritance occurs when multiple classes inherit from a single superclass. All subclasses share the common superclass.

Example

```
class Animal {
  void eat() {
    System.out.println("Animal eats");
  }
}

class Dog extends Animal {
  void bark() {
```

```
    System.out.println("Dog barks");
  }
}

class Cat extends Animal {
  void meow() {
    System.out.println("Cat meows");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog myDog = new Dog();
    myDog.eat(); // From Animal class
    myDog.bark(); // Dog class method

    Cat myCat = new Cat();
    myCat.eat(); // From Animal class
    myCat.meow(); // Cat class method
  }
}
```

4. Multiple Inheritance (through Interfaces)

Multiple Inheritance is not directly supported in Java with classes due to the ambiguity problem (the Diamond Problem). However, Java allows multiple inheritance through interfaces. A class can implement multiple interfaces, thereby inheriting behaviors from multiple sources.

Example

```
interface Animal {
  void eat();
}

interface Pet {
  void play();
}

class Dog implements Animal, Pet {
  public void eat() {
    System.out.println("Dog eats");
  }

  public void play() {
    System.out.println("Dog plays");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog myDog = new Dog();
    myDog.eat(); // From Animal interface
```

```
    myDog.play(); // From Pet interface
  }
}
```

5. Hybrid Inheritance (not directly supported)

Hybrid Inheritance is a combination of two or more types of inheritance. Java does not support hybrid inheritance with classes to avoid complexity and ambiguity issues. However, hybrid inheritance can be achieved using interfaces in combination with other forms of inheritance.

Example

```
interface Animal {
  void eat();
}

class Mammal implements Animal {
  public void eat() {
    System.out.println("Mammal eats");
  }
}

class Dog extends Mammal {
  void bark() {
    System.out.println("Dog barks");
  }
}

class Cat extends Mammal {
  void meow() {
    System.out.println("Cat meows");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog myDog = new Dog();
    myDog.eat(); // From Mammal class
    myDog.bark(); // Own method

    Cat myCat = new Cat();
    myCat.eat(); // From Mammal class
    myCat.meow(); // Cat class method
  }
}
```

Summary

- Single Inheritance: A class inherits from one superclass.
- Multilevel Inheritance: A class inherits from another derived class, creating a chain.
- Hierarchical Inheritance: Multiple classes inherit from a single superclass.
- Multiple Inheritance (through Interfaces): A class implements multiple interfaces to inherit multiple behaviors.

- Hybrid Inheritance: Java does not support hybrid inheritance with classes but can achieve it using interfaces.

16. Polymorphism

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP) in Java. It allows objects of dif erent classes to be treated as objects of a common superclass, primarily through method overriding and method overloading. Polymorphism enables a single action to behave dif erently based on the object that performs the action.

Types of Polymorphism in Java

1. Compile-Time Polymorphism (Method Overloading)
2. Runtime Polymorphism (Method Overriding)

1. Compile-Time Polymorphism (Method Overloading)
Method Overloading allows multiple methods in the same class to have the same name but dif erent parameter lists (dif erent type, number, or both). The method that gets called is determined at compile time based on the method signature and the arguments passed to it.

Example of Method Overloading

```
class Printer {
    // Method with one parameter
    void print(int i) {
        System.out.println("Printing integer: " + i);
    }

    // Method with two parameters
    void print(String s, int i) {
        System.out.println("Printing string and integer: " + s + ", " + i);
    }

    // Method with dif erent parameter types
    void print(double d) {
        System.out.println("Printing double: " + d);
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.print(10);          // Calls print(int)
        printer.print("Number", 20); // Calls print(String, int)
        printer.print(15.5);        // Calls print(double)
    }
}
```

Output:

```
Printing integer: 10
Printing string and integer: Number, 20
```

```
Printing double: 15.5
```

2. Runtime Polymorphism (Method Overriding)
Method Overriding occurs when a subclass provides a specif c implementation of a method that is already def ned in its superclass. The method in the subclass should have the same name, return type, and parameter list as the method in the superclass. The method to be executed is determined at runtime based on the object's actual class.

Example of Method Overriding

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

class Dog extends Animal {
   @Override
   void makeSound() {
      System.out.println("Dog barks");
   }
}

class Cat extends Animal {
   @Override
   void makeSound() {
      System.out.println("Cat meows");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal myDog = new Dog();
      Animal myCat = new Cat();

      myDog.makeSound(); // Calls Dog's makeSound method
      myCat.makeSound(); // Calls Cat's makeSound method
   }
}
```

Output:

```
Dog barks
Cat meows
```

Key Concepts of Polymorphism
- Method Overloading:
  - Compile-time Polymorphism: Method resolution is done during compile time based on method signatures (name and parameters).
  - Signature: Overloaded methods must dif er in the number or type of parameters.

- Method Overriding:

- Runtime Polymorphism: Method resolution is done during runtime based on the object's actual class.
- Inheritance: The subclass method overrides the superclass method.
- Dynamic Method Dispatch: The Java runtime determines which method implementation to execute based on the object's type at runtime.

Benef ts of Polymorphism

- Code Reusability: Allows using the same method name for dif erent implementations, making the code easier to understand and maintain.
- Flexibility and Extensibility: New classes can be introduced with minimal changes to existing code.
- Dynamic Behavior: Enables objects to be handled more generically, allowing dif erent implementations to be swapped out dynamically.

Summary

- Compile-Time Polymorphism: Achieved through method overloading, where multiple methods with the same name have dif erent parameters.
- Runtime Polymorphism: Achieved through method overriding, where a subclass provides a specif c implementation for a method def ned in the superclass.

17. Access Modif ers

Access modif ers are keywords used to def ne the visibility or accessibility of classes, methods, f elds, and constructors. They control which parts of a program can access these elements. Java provides four primary access modif ers, each with its own level of access control:

1. public

- Visibility: Accessible from any other class or package.
- Use: When you want to allow unrestricted access to a class, method, f eld, or constructor.

Example:

```
public class PublicClass {
   public int publicField;

   public void publicMethod() {
      System.out.println("This is a public method");
   }
}
```

2. protected

- Visibility: Accessible within the same package and by subclasses (even if they are in dif erent packages).
- Use: When you want to allow access to a class, method, or f eld within the same package and by subclasses, but not necessarily by other classes.

Example:

```
class ProtectedClass {
   protected int protectedField;
```

```
    protected void protectedMethod() {
        System.out.println("This is a protected method");
    }
}

class SubClass extends ProtectedClass {
    void accessProtectedMembers() {
        protectedField = 10; // Accessing protected f eld
        protectedMethod(); // Accessing protected method
    }
}
```

3. default (Package-Private)
- Visibility: Accessible only within the same package. If no access modif er is specif ed, the default access level is applied.
- Use: When you want to restrict access to within the same package, but not to subclasses in other packages or other classes.

Example:

```
class DefaultClass {
    int defaultField; // Package-private f eld

    void defaultMethod() {
        System.out.println("This is a default method");
    }
}
```

4. private
- Visibility: Accessible only within the same class. Not accessible from any other class, even subclasses.
- Use: When you want to restrict access to a class's internal details, ensuring encapsulation and hiding implementation details.

Example:

```
class PrivateClass {
    private int privateField;

    private void privateMethod() {
        System.out.println("This is a private method");
    }

    void accessPrivateMembers() {
        privateField = 20; // Accessing private f eld within the same class
        privateMethod(); // Accessing private method within the same class
    }
}
```

Summary
- 'public': Accessible from any other class or package. No restriction on visibility.

- 'protected': Accessible within the same package and by subclasses (even if they are in dif erent packages).
- 'default' (Package-Private): Accessible only within the same package.
- 'private': Accessible only within the same class. Restricts access to the class itself.

Access Modif er Usage in Classes
- Class: Only 'public' and default access modif ers are applicable to classes. A class must be 'public' if it is def ned in a f le with the same name as the class. A class with no access modif er is package-private.
- Fields and Methods: All four access modif ers can be applied to f elds and methods to control their visibility.

18. Exception Handling

Exception Handling is a mechanism to handle runtime errors, allowing a program to continue its execution or fail gracefully. Exceptions are events that disrupt the normal f ow of the program and need to be handled to avoid abrupt termination.

Key Concepts in Exception Handling

1. Exception:
- An object that represents an error or unexpected event that occurs during program execution. Examples include division by zero, f le not found, etc.

2. Exception Handling Blocks:
- 'try': Contains code that might throw an exception.
- 'catch': Catches and handles exceptions thrown by the 'try' block.
- 'f nally': Contains code that is always executed, regardless of whether an exception was thrown or not.
- 'throw': Used to explicitly throw an exception.
- 'throws': Indicates that a method might throw an exception and must be handled by the caller.

Basic Syntax

```
try {
   // Code that might throw an exception
} catch (ExceptionType e) {
   // Code to handle the exception
} f nally {
   // Code that always executes, whether an exception occurred or not
}
```

Example of Exception Handling

```
public class ExceptionHandlingExample {
   public static void main(String[] args) {
      int[] numbers = {1, 2, 3};

      try {
         // Code that may throw an exception
         int result = numbers[3]; // ArrayIndexOutOfBoundsException
```

```
        System.out.println("Result: " + result);
    } catch (ArrayIndexOutOfBoundsException e) {
        // Handling the exception
        System.out.println("Error: Array index out of bounds!");
    } f nally {
        // This block always executes
        System.out.println("Finally block executed.");
    }

    System.out.println("Program continues...");
  }
}
```

Output:

```
Error: Array index out of bounds!
Finally block executed.
Program continues...
```

1. 'try' Block:
   - Contains code that might throw an exception. In this example, accessing 'numbers[3]' causes an 'ArrayIndexOutOfBoundsException' because the array has only 3 elements (indices 0 to 2).

2. 'catch' Block:
   - Catches the exception and handles it. In this case, it prints an error message when the exception is caught.

3. 'f nally' Block:
   - Executes regardless of whether an exception was thrown or not. It is typically used to release resources like closing f les or network connections.

Throwing Exceptions

You can throw exceptions explicitly using the 'throw' keyword.

```
public class ThrowExample {
  public static void checkNumber(int number) {
    if (number < 0) {
      throw new IllegalArgumentException("Number must be positive");
    }
    System.out.println("Number is: " + number);
  }

  public static void main(String[] args) {
    try {
      checkNumber(-1); // Throws an exception
    } catch (IllegalArgumentException e) {
      System.out.println("Caught exception: " + e.getMessage());
    }
  }
}
```

Declaring Exceptions
Methods can declare exceptions they might throw using the 'throws' keyword.

```
public class ThrowsExample {
   public static void riskyMethod() throws Exception {
      throw new Exception("This is an exception");
   }

   public static void main(String[] args) {
      try {
         riskyMethod();
      } catch (Exception e) {
         System.out.println("Caught exception: " + e.getMessage());
      }
   }
}
```

Types of Exceptions

1. Checked Exceptions:
- Must be either caught or declared in the method signature using 'throws'. Examples include 'IOException', 'SQLException'.

2. Unchecked Exceptions:
- Also known as runtime exceptions, these do not need to be caught or declared. Examples include 'ArithmeticException', 'ArrayIndexOutOfBoundsException', 'NullPointerException'.

Summary
- Exception Handling: Allows a program to manage errors and continue running.
- 'try' Block: Contains code that might throw an exception.
- 'catch' Block: Handles the exception if it occurs.
- 'f nally' Block: Always executes, regardless of exceptions.
- 'throw' Keyword: Used to explicitly throw an exception.
- 'throws' Keyword: Indicates that a method might throw an exception.
- Checked vs. Unchecked Exceptions: Checked exceptions must be handled or declared, while unchecked exceptions do not.

19. Collections Framework

The Collections Framework in Java provides a set of interfaces, classes, and algorithms that facilitate the manipulation and management of groups of objects. It of ers a standardized way to handle collections of data, such as lists, sets, and maps, and provides a range of powerful data structures and utilities to work with them.

Key Components of the Collections Framework

1. Interfaces
2. Implementations
3. Algorithms
4. Utility Classes

1. Interfaces
The core interfaces of the Collections Framework def ne various types of collections. Here are some of the main ones:

- 'Collection': The root interface of the collection hierarchy. It represents a group of objects known as elements. It has subinterfaces like 'List', 'Set', and 'Queue'.

- 'List': An ordered collection that allows duplicate elements. It provides positional access to elements. Common implementations include 'ArrayList', 'LinkedList', and 'Vector'.

- 'Set': A collection that does not allow duplicate elements. Common implementations include 'HashSet', 'LinkedHashSet', and 'TreeSet'.

- 'Queue': A collection designed for holding elements prior to processing. It typically represents a data structure where elements are added and removed according to specif c rules (e.g., FIFO - First In, First Out). Common implementations include 'PriorityQueue' and 'LinkedList'.

- 'Map': An object that maps keys to values. It does not inherit from the 'Collection' interface. Common implementations include 'HashMap', 'LinkedHashMap', and 'TreeMap'.

2. Implementations
Each interface in the Collections Framework has one or more concrete implementations that provide the actual data structure and behavior. Here are some key implementations:

- 'ArrayList': Implements the 'List' interface. It uses a dynamic array to store elements and provides fast random access.

- 'LinkedList': Implements both 'List' and 'Deque' interfaces. It uses a doubly linked list and is ef icient for insertions and deletions.

- 'HashSet': Implements the 'Set' interface using a hash table. It does not guarantee any specif c order of elements.

- 'TreeSet': Implements the 'Set' interface using a red-black tree. It provides a sorted order of elements.

- 'HashMap': Implements the 'Map' interface using a hash table. It allows null values and keys and does not guarantee order.

- 'TreeMap': Implements the 'Map' interface using a red-black tree. It provides a sorted order of keys.

3. Algorithms
The Collections Framework provides a set of algorithms that can be applied to collections. These include sorting, searching, and shuf ling.

- 'Collections.sort(List<T> list)': Sorts the specif ed list into ascending order.

- 'Collections.binarySearch(List<? extends Comparable<? super T>> list, T key)': Searches for the specif ed object in the sorted list.

- 'Collections.shuf le(List<?> list)': Randomly permutes the elements in the list.

4. Utility Classes
The framework includes utility classes that provide static methods to operate on or return collections.

- 'Collections': A utility class with static methods for operating on collections, such as sorting, reversing, and f nding the maximum or minimum element.

- 'Arrays': A utility class for manipulating arrays, including methods to convert arrays to lists and vice versa.

Here's a brief example demonstrating the use of some collection types:

```
import java.util.*;

public class CollectionsExample {
    public static void main(String[] args) {
        // Using ArrayList
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");
        Collections.sort(list); // Sorting the list
        System.out.println("Sorted List: " + list);

        // Using HashSet
        Set<String> set = new HashSet<>();
        set.add("Dog");
        set.add("Cat");
        set.add("Bird");
        System.out.println("Set: " + set);

        // Using HashMap
        Map<String, Integer> map = new HashMap<>();
        map.put("John", 25);
        map.put("Alice", 30);
        map.put("Bob", 20);
        System.out.println("Map: " + map);
    }
}
```

Output:

```
Sorted List: [Apple, Banana, Cherry]
Set: [Dog, Cat, Bird]
Map: {John=25, Alice=30, Bob=20}
```

Summary
- Interfaces: Def ne the core types of collections ('Collection', 'List', 'Set', 'Queue', 'Map').
- Implementations: Provide concrete data structures ('ArrayList', 'HashSet', 'HashMap', etc.).

- Algorithms: Provide operations like sorting and searching.
- Utility Classes: Of er static methods for common operations ('Collections', 'Arrays').

20. Sample Programs

1. Simple Calculator

This program demonstrates basic arithmetic operations (addition, subtraction, multiplication, and division) using user input.

```
import java.util.Scanner;

public class SimpleCalculator {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Enter f rst number: ");
    double num1 = scanner.nextDouble();

    System.out.print("Enter second number: ");
    double num2 = scanner.nextDouble();

    System.out.print("Enter operation (+, -, *, /): ");
    char operation = scanner.next().charAt(0);

    double result;

    switch (operation) {
      case '+':
        result = num1 + num2;
        break;
      case '-':
        result = num1 - num2;
        break;
      case '*':
        result = num1 * num2;
        break;
      case '/':
        if (num2 != 0) {
          result = num1 / num2;
        } else {
          System.out.println("Error: Division by zero.");
          return;
        }
        break;
      default:
        System.out.println("Invalid operation.");
        return;
```

```
    }

    System.out.println("Result: " + result);
  }
}
```

2. Factorial Calculation
This program calculates the factorial of a number using a loop.

```
import java.util.Scanner;

public class FactorialCalculator {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Enter a number: ");
    int number = scanner.nextInt();

    int factorial = 1;
    for (int i = 1; i <= number; i++) {
      factorial *= i;
    }

    System.out.println("Factorial of " + number + " is " + factorial);
  }
}
```

3. Array Operations
This program demonstrates basic operations on arrays, including initializing, iterating, and calculating the sum of array elements.

```
public class ArrayOperations {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};
    int sum = 0;

    System.out.println("Array elements:");
    for (int num : numbers) {
      System.out.println(num);
      sum += num;
    }

    System.out.println("Sum of array elements: " + sum);
  }
}
```

4. String Manipulation
This program shows how to manipulate strings, including concatenation, f nding the length, and converting to uppercase.

```
public class StringManipulation {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";

        // Concatenation
        String concatenated = str1 + " " + str2;
        System.out.println("Concatenated String: " + concatenated);

        // Length
        int length = concatenated.length();
        System.out.println("Length of concatenated string: " + length);

        // Convert to uppercase
        String upperCase = concatenated.toUpperCase();
        System.out.println("Uppercase String: " + upperCase);
    }
}
```

5. Basic Class and Object

This program demonstrates how to def ne a class, create objects, and access class members.

```
// Def ne a class with f elds and methods
class Car {
    String brand;
    int year;

    // Method to display car details
    void displayDetails() {
        System.out.println("Brand: " + brand);
        System.out.println("Year: " + year);
    }
}

public class CarExample {
    public static void main(String[] args) {
        // Create an object of the Car class
        Car myCar = new Car();
        myCar.brand = "Toyota";
        myCar.year = 2020;

        // Call the method on the object
        myCar.displayDetails();
    }
}
```

6. Inheritance

This program demonstrates inheritance, where a `Vehicle` class is extended by the `Bike` class.

```
// Base class
class Vehicle {
    String type = "Vehicle";

    void displayType() {
        System.out.println("Type: " + type);
    }
}

// Derived class
class Bike extends Vehicle {
    String model = "Bike";

    void displayModel() {
        System.out.println("Model: " + model);
    }
}

public class InheritanceExample {
    public static void main(String[] args) {
        // Create an object of the derived class
        Bike myBike = new Bike();
        myBike.displayType(); // Inherited method
        myBike.displayModel(); // Method in Bike class
    }
}
```

7. Polymorphism with Method Overloading

This program demonstrates method overloading, where multiple methods have the same name but dif erent parameters.

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

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

    // Overloaded method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}

public class PolymorphismExample {
```

```
    public static void main(String[] args) {
        MathOperations math = new MathOperations();

        // Call overloaded methods
        System.out.println("Sum of 5 and 10: " + math.add(5, 10));
        System.out.println("Sum of 1, 2, and 3: " + math.add(1, 2, 3));
        System.out.println("Sum of 5.5 and 10.2: " + math.add(5.5, 10.2));
    }
}
```

8. Polymorphism with Method Overriding

This program demonstrates method overriding, where a derived class provides a specif c implementation of a method already def ned in its base class.

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

// Derived class
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

// Another derived class
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class PolymorphismOverridingExample {
    public static void main(String[] args) {
        Animal myAnimal;

        // Reference to base class object
        myAnimal = new Dog();
        myAnimal.makeSound(); // Calls Dog's makeSound

        myAnimal = new Cat();
        myAnimal.makeSound(); // Calls Cat's makeSound
    }
}
```

9. Inheritance and Polymorphism Combined

This program demonstrates both inheritance and polymorphism, where dif erent types of shapes are handled using a common base class.

```
// Base class
abstract class Shape {
    abstract void draw();
}

// Derived class
class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

// Another derived class
class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a Rectangle");
    }
}

public class ShapeTest {
    public static void main(String[] args) {
        Shape shape;

        // Reference to base class object
        shape = new Circle();
        shape.draw(); // Calls Circle's draw

        shape = new Rectangle();
        shape.draw(); // Calls Rectangle's draw
    }
}
```

10. Exception Handling

This program demonstrates how to handle exceptions using try, catch, and f nally blocks.

```
import java.util.Scanner;

public class ExceptionHandlingExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter a number: ");
            int number = scanner.nextInt();
            int result = 10 / number;
            System.out.println("Result: " + result);
```

```
    } catch (ArithmeticException e) {
        System.out.println("Error: Division by zero is not allowed.");
    } catch (java.util.InputMismatchException e) {
        System.out.println("Error: Invalid input. Please enter a number.");
    } f nally {
        System.out.println("Execution f nished.");
        scanner.close();
    }
  }
}
```