Unit 1: Introduction to Java Programming Language

Java Overview

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It was originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform.

Brief History & Evolution of Java

- **1991**: The project that led to Java started with the aim of developing software for various types of consumer electronic devices. The language was initially called "Oak" after an oak tree outside Gosling's office.
- **1995**: Renamed as "Java", the language was officially released. It was designed to be platform-independent and secure, which was achieved by compiling to bytecode that runs on a Java Virtual Machine (JVM).
- **1998**: Java 2 was released as J2SE (Java 2 Platform, Standard Edition), J2EE (Java 2 Platform, Enterprise Edition), and J2ME (Java 2 Platform, Micro Edition) for different application environments.
- **2004-2006**: Java 5 (Tiger) introduced significant language features such as generics, metadata, and enhanced for-loop.
- **2010**: Oracle Corporation acquired Sun Microsystems and took over the stewardship of Java.
- **2011-present**: Java has continued to evolve with a faster release cadence introduced by Oracle, with new versions released every six months to bring developers new features and updates more quickly.

Java Features

- **Platform Independent**: Java code is compiled into bytecode, which can run on any device equipped with a JVM, enabling the famous principle of "write once, run anywhere" (WORA).
- **Object-Oriented**: Java strictly follows the object-oriented programming model, including concepts like inheritance, encapsulation, polymorphism, and abstraction.
- **Robust and Secure**: Java offers strong memory management, exception handling, and type-checking mechanisms. Its security features include the sandbox environment of the JVM.
- **Multithreaded**: Java supports multithreaded programming, allowing developers to build applications that can perform multiple tasks simultaneously.
- **Rich API**: Java provides a comprehensive standard library (API) that includes tools for networking, I/O, data structures, concurrency, and more.
- **High Performance**: While the early versions were criticized for performance, Java has significantly improved with the introduction of Just-In-Time (JIT) compilation and various optimization techniques.

Java Applications

- **Desktop Applications**: Java is used to develop cross-platform desktop applications. Swing and JavaFX are notable APIs for creating rich graphical user interfaces.
- **Web Applications**: Java is widely used in web development, with technologies such as Servlets, JSPs (JavaServer Pages), and frameworks like Spring and Hibernate facilitating the development of robust web applications.
- **Mobile Applications**: Java was the official language for Android app development until the introduction of Kotlin as an alternative. It remains widely used for Android development.
- **Enterprise Applications**: Java EE (Enterprise Edition) provides APIs and runtime environments for developing and running large-scale, multi-tiered, scalable, and secure network applications.

Java's versatility, robustness, and widespread adoption have cemented its place as a cornerstone of modern software development, covering a wide array of computing platforms from embedded devices to enterprise servers and supercomputers.

Java Environment Setup & Basic Java Syntax

Java Components

- **JVM (Java Virtual Machine)**: JVM is an abstract computing machine that enables Java bytecode to be executed on different platforms. It interprets the bytecode into machine-specific instructions.
- **JRE (Java Runtime Environment)**: JRE includes JVM along with libraries and other components required to run Java applications but does not include development tools.
- **JDK (Java Development Kit)**: JDK is a full-featured software development kit that includes JRE, compilers, debuggers, and other tools necessary for developing Java applications.

Setting up Java Development Environment

To set up a Java development environment:

1. **Download JDK**: Visit the official Oracle website or adopt openJDK distributions and download the JDK appropriate for your operating system.
2. **Install JDK**: Follow the installation instructions provided by Oracle or the respective distribution. This usually involves running an installer program.
3. **Set up Environment Variables**: Set the `JAVA_HOME` environment variable to point to the JDK installation directory and add the JDK's `bin` directory to the `PATH` environment variable.
4. **Verify Installation**: Open a command prompt or terminal and type `java -version` and `javac -version` to verify that Java and the Java compiler are installed correctly.

Structure of a Java Program

A basic Java program consists of:

- **Class Declaration**: Every Java program begins with a class declaration. The class name should match the filename.
- **Main Method**: The main method is the entry point of a Java program. It has the following syntax:

```
public static void main(String[] args) {
    // Program logic goes here
}
```

- **Output in Java**: Output in Java is typically achieved using the `System.out.println()` method:

  ```
  System.out.println("Hello, World!");
  ```

- **Comments**: Java supports single-line (`//`) and multi-line (`/* */`) comments for documenting code.

Compilation and Execution of Java Program

To compile and execute a Java program:

1. **Write Code**: Create a Java source file with the `.java` extension containing the Java code.
2. **Compile Code**: Open a terminal or command prompt, navigate to the directory containing the Java file, and use the `javac` command to compile the code:

   ```
   javac YourProgram.java
   ```

3. **Execute Program**: After successfully compiling, use the `java` command followed by the name of the class containing the main method (without the `.class` extension) to execute the program:

   ```
   java YourProgram
   ```

Importance of Bytecode & Garbage Collection

- **Bytecode**: Java source code is compiled into bytecode, which is a platform-independent intermediate representation. This bytecode can be executed on any device with a JVM, enabling Java's "write once, run anywhere" capability.
- **Garbage Collection**: Java employs automatic memory management through garbage collection. It automatically deallocates memory occupied by objects that are no longer in use, preventing memory leaks and simplifying memory management for developers. Garbage collection helps ensure memory efficiency and program stability in Java applications.

Data Types, Identifiers, Constants, and Variables

Primitive Data Types

Java supports eight primitive data types:

- **byte**: 8-bit signed integer
- **short**: 16-bit signed integer
- **int**: 32-bit signed integer
- **long**: 64-bit signed integer
- **float**: 32-bit floating-point number
- **double**: 64-bit floating-point number

- **char**: 16-bit Unicode character
- **boolean**: Represents true or false values

Type Conversion and Casting

- **Implicit Conversion**: Java automatically converts smaller data types to larger ones to prevent loss of data. For example, `int` can be implicitly converted to `long`.
- **Explicit Conversion (Casting)**: When converting larger data types to smaller ones, explicit casting is required to avoid loss of data. For example:

```
double d = 10.5;
int i = (int) d; // Explicit casting
```

Identifiers and Naming Conventions

- **Identifiers**: Identifiers are names given to classes, methods, variables, etc., in Java. They must start with a letter, underscore (_), or dollar sign ($), followed by letters, digits, underscores, or dollar signs.
- **Naming Conventions**:
  - Class names should start with an uppercase letter and follow CamelCase (e.g., `MyClass`).
  - Variable and method names should start with a lowercase letter and follow camelCase (e.g., `myVariable`, `myMethod`).
  - Constants should be all uppercase with underscores separating words (e.g., `MAX_SIZE`).

Variable Declaration and Initialization

- **Variable Declaration**: Variables are declared with a data type followed by a name:

```
int myVariable;
```

- **Variable Initialization**: Variables can be initialized at the time of declaration or later in the code:

```
int myVariable = 10; // Initialization at declaration
myVariable = 20;     // Later initialization
```

Scope of Variables

- **Instance Variables**: Variables declared within a class but outside any method are instance variables. They exist as long as the object they belong to exists.
- **Local Variables**: Variables declared within a method or block have local scope. They exist only within the method or block where they are declared.
- **Class Variables (Static Variables)**: Variables declared with the `static` keyword within a class are class variables. They are shared among all instances of the class.

Declaring Constants (Final Keyword)

- Constants in Java are declared using the `final` keyword. Once assigned a value, a final variable cannot be reassigned.

```
final double PI = 3.14159;
```

- By convention, constant names are written in uppercase letters with underscores separating words.

Arrays

One-dimensional Arrays

- **Declaration**: To declare a one-dimensional array, specify the type of elements followed by square brackets []:

```
int[] numbers;
```

- **Initialization**: Arrays can be initialized using the `new` keyword followed by the type and the number of elements, or directly with values enclosed in curly braces {}:

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

- **Accessing Elements**: Elements of an array are accessed using the index (starting from 0):

```
int[] numbers = {1, 2, 3, 4, 5};
int firstElement = numbers[0]; // Accessing first element
```

Multidimensional Arrays (Two-dimensional)

- **Declaration**: To declare a two-dimensional array, specify the type of elements followed by two sets of square brackets [][]:

```
int[][] matrix;
```

- **Initialization**: Two-dimensional arrays can be initialized similarly to one-dimensional arrays, with each row enclosed in curly braces {}:

```
int[][] matrix = new int[3][3]; // Initializing with size
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // Initializing with
values
```

- **Accessing Elements**: Elements of a two-dimensional array are accessed using row and column indices:

```
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int element = matrix[1][2]; // Accessing element at row 1, column 2 (value:
6)
```

- **Iterating Through a Two-dimensional Array**: Nested loops are commonly used to iterate through all elements of a two-dimensional array:

```java
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        // Accessing each element using matrix[i][j]
    }
}
```

Two-dimensional arrays can represent matrices, tables, grids, etc., and are useful for storing and processing structured data in Java.

Operators

Arithmetic Operators (+, -, *, /, %)

Arithmetic operators are used to perform mathematical operations.

- **Example**:

```java
int a = 10;
int b = 3;
int sum = a + b;        // Addition
int difference = a - b; // Subtraction
int product = a * b;    // Multiplication
int quotient = a / b;   // Division
int remainder = a % b;  // Modulus (remainder)
```

Relational Operators (>, <, >=, <=, ==, !=)

Relational operators are used to establish relationships between two values.

- **Example**:

```java
int a = 10;
int b = 5;
boolean greater = a > b;
boolean lesserOrEqual = a <= b;
boolean isEqual = a == b;
boolean notEqual = a != b;
```

Logical Operators (&&, ||, !)

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

- **Example**:

```java
boolean x = true;
boolean y = false;
boolean result1 = x && y; // Logical AND
boolean result2 = x || y; // Logical OR
boolean result3 = !x;     // Logical NOT (negation)
```

Bitwise Operators

Bitwise operators perform bitwise operations on integer operands.

- **Example**:

```
int a = 5;  // Binary: 101
int b = 3;  // Binary: 011
int bitwiseAnd = a & b;       // Bitwise AND
int bitwiseOr = a | b;        // Bitwise OR
int bitwiseXor = a ^ b;       // Bitwise XOR
int bitwiseComplement = ~a;   // Bitwise complement
int leftShift = a << 1;       // Left shift
int rightShift = a >> 1;      // Right shift
```

Assignment Operators (=, +=, -=, etc.)

Assignment operators are used to assign values to variables.

- **Example**:

```
int a = 10;
a += 5; // Equivalent to a = a + 5;
```

Increment/Decrement Operators (++ ,--)

Increment and decrement operators are used to increase or decrease the value of a variable by 1.

- **Example**:

```
int a = 5;
a++; // Increment a by 1
int b = 10;
b--; // Decrement b by 1
```

Conditional (Ternary) Operator(?:)

The conditional operator is a ternary operator that evaluates a boolean expression and returns one of two values depending on whether the expression is true or false.

- **Example**:

```
int a = 10;
int b = 5;
int max = (a > b) ? a : b; // max will be assigned the value of a if a is
greater than b, otherwise b
```

These operators are fundamental in Java for performing various operations and making decisions based on conditions.

Control Flow Statements

Control flow statements in Java are used to control the flow of execution in a program based on certain conditions or loops.

Selection Statements (if, if-else, switch-case)

- **if Statement**: Executes a block of code if a specified condition is true.

```java
int x = 10;
if (x > 5) {
    System.out.println("x is greater than 5");
}
```

- **if-else Statement**: Executes one block of code if the specified condition is true and another block of code if it's false.

```java
int x = 10;
if (x > 5) {
    System.out.println("x is greater than 5");
} else {
    System.out.println("x is less than or equal to 5");
}
```

- **switch-case Statement**: Evaluates an expression and executes code blocks based on matching cases.

```java
int day = 3;
switch (day) {
    case 1:
        System.out.println("Sunday");
        break;
    case 2:
        System.out.println("Monday");
        break;
    // Other cases...
    default:
        System.out.println("Invalid day");
}
```

Looping Statements (for, while, do-while)

- **for Loop**: Executes a block of code a specified number of times.

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

- **while Loop**: Executes a block of code repeatedly as long as a specified condition is true.

```java
int i = 0;
while (i < 5) {
    System.out.println("Iteration: " + i);
    i++;
}
```

- **do-while Loop**: Similar to the while loop, but the block of code is executed at least once before the condition is checked.

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

Jump Statements (break, continue, return)

- **break Statement**: Terminates the loop or switch statement and transfers control to the statement immediately following the loop or switch.

```java
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        break; // Terminates the loop when i equals 3
    }
    System.out.println("Iteration: " + i);
}
```

- **continue Statement**: Skips the current iteration of a loop and proceeds with the next iteration.

```java
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue; // Skips iteration when i equals 3
    }
    System.out.println("Iteration: " + i);
}
```

- **return Statement**: Exits the current method and returns a value (if applicable) to the caller.

```java
public int add(int a, int b) {
    return a + b; // Returns the sum of a and b
}
```

These control flow statements provide essential mechanisms for directing the flow of execution in Java programs, allowing developers to implement conditional logic and repetitive tasks efficiently.

Unit 2: Object-Oriented Programming

Procedure-Oriented vs. Object-Oriented Programming

Characteristics

Procedure-Oriented Programming (POP):

1. **Focus**: POP focuses on functions or procedures that operate on data.
2. **Data and Functions**: Data and functions are separate entities.
3. **Global Data**: Relies heavily on global data, which can lead to data integrity issues.
4. **Procedural Abstraction**: Emphasizes procedural abstraction, breaking down a problem into a sequence of steps.
5. **Top-Down Approach**: Follows a top-down approach, where the problem is broken down into smaller sub-problems.

Object-Oriented Programming (OOP):

1. **Focus**: OOP focuses on objects that encapsulate data and behavior.
2. **Data Encapsulation**: Data and functions are encapsulated within objects, promoting data hiding and encapsulation.
3. **Class and Object**: Relies on classes and objects to model real-world entities and interactions.
4. **Inheritance and Polymorphism**: Supports inheritance and polymorphism, enabling code reuse and flexibility.
5. **Bottom-Up Approach**: Often follows a bottom-up approach, where objects are identified and modeled to represent real-world entities.

Differences

1. **Data Abstraction**:
   - POP: Relies on procedural abstraction, focusing on procedures or functions.
   - OOP: Uses data abstraction through classes and objects, encapsulating data and behavior into objects.
2. **Data Hiding**:
   - POP: Doesn't inherently support data hiding; global data is accessible from anywhere in the program.
   - OOP: Supports data hiding through encapsulation, allowing access to data only through well-defined interfaces.
3. **Code Reusability**:
   - POP: Code reuse is limited, often leading to code duplication.
   - OOP: Encourages code reuse through inheritance and composition, facilitating modular and extensible designs.
4. **Maintenance**:
   - POP: Maintenance can be challenging due to the lack of encapsulation and modularity.
   - OOP: Offers better maintenance capabilities with encapsulation and modular design principles.

5. **Flexibility**:
   - POP: Can be less flexible when requirements change, as changes might require modifications across multiple functions.
   - OOP: Offers greater flexibility with features like polymorphism and dynamic binding, allowing changes to be localized within objects.
6. **Complexity**:
   - POP: Generally simpler in structure and easier to understand for small-scale projects.
   - OOP: Can be more complex due to the added concepts of classes, objects, and inheritance, but provides better scalability and organization for large-scale projects.

In summary, while POP emphasizes procedures and functions, OOP revolves around objects and their interactions, offering better encapsulation, code reusability, and maintainability for complex software systems. The choice between them often depends on the nature and scale of the project, as well as the preferences of the development team.

OOP Concepts

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which encapsulate data and behavior. OOP provides several key concepts to facilitate modular and organized software design.

1. Classes and Objects

- **Class**: A class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of the class will have.

```
public class Car {
    String color;
    int speed;

    void accelerate() {
        // Method to increase speed
    }

    void brake() {
        // Method to decrease speed
    }
}
```

- **Object**: An object is an instance of a class. It represents a real-world entity and encapsulates both data (attributes) and behavior (methods).

```
Car myCar = new Car();
myCar.color = "Red";
myCar.speed = 60;
myCar.accelerate();
```

2. Encapsulation

Encapsulation is the bundling of data (attributes) and methods (behaviors) that operate on the data into a single unit (object). It helps in hiding the internal state of an object and protecting it from external interference.

- **Example**:

```java
public class BankAccount {
    private double balance;

    public void deposit(double amount) {
        // Method to deposit money
    }

    public void withdraw(double amount) {
        // Method to withdraw money
    }
}
```

3. Abstraction

Abstraction refers to the process of hiding the implementation details of a class and showing only the essential features to the outside world. It focuses on what an object does rather than how it does it.

- **Example**:

```java
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        // Method to draw a circle
    }
}

class Rectangle implements Shape {
    public void draw() {
        // Method to draw a rectangle
    }
}
```

4. Inheritance

Inheritance is a mechanism in which a new class (derived class or subclass) inherits properties and behaviors from an existing class (base class or superclass). It promotes code reuse and establishes a hierarchical relationship between classes.

- **Example**:

```java
class Animal {
    void eat() {
        // Method to eat
    }
}

class Dog extends Animal {
    void bark() {
        // Method to bark
    }
}
```

5. Polymorphism

Polymorphism allows objects to be treated as instances of their superclass or as instances of their subclass. It enables flexibility and dynamic behavior in the program.

- **Example**:

```java
class Animal {
    void makeSound() {
        // Method to make a generic animal sound
    }
}

class Dog extends Animal {
    void makeSound() {
        // Method to make a dog sound
    }
}

class Cat extends Animal {
    void makeSound() {
        // Method to make a cat sound
    }
}
```

These OOP concepts form the foundation of object-oriented design and programming. They enable developers to create modular, maintainable, and scalable software systems by modeling real-world entities and interactions in a structured and organized manner.

Classes and Objects

Defining Classes

In Java, a class is a blueprint for creating objects. It defines the structure and behavior of objects of that type.

```java
public class MyClass {
    // Class body
}
```

Attributes and Methods

- **Attributes**: Attributes are variables that define the state of objects. They represent the data associated with objects of the class.

```
public class Car {
    String color;
    int speed;
}
```

- **Methods**: Methods are functions that define the behavior of objects. They represent the actions that objects of the class can perform.

```
public class Car {
    void accelerate() {
        // Method to increase speed
    }

    void brake() {
        // Method to decrease speed
    }
}
```

Creating Objects

Objects are instances of classes. They are created using the `new` keyword followed by the class constructor.

```
MyClass obj = new MyClass();
```

- **this Keyword**: Inside a method or constructor, `this` refers to the current object. It is used to differentiate between instance variables and local variables with the same name.

```
public class Person {
    String name;

    public void setName(String name) {
        this.name = name; // Assigning the parameter value to the instance
variable
    }
}
```

Access Modifiers

Access modifiers control the visibility of classes, attributes, methods, and constructors.

- **public**: Accessible from anywhere.
- **protected**: Accessible within the same package and subclasses (even if they are in different packages).
- **private**: Accessible only within the same class.
- **default (no modifier)**: Accessible within the same package.

```
public class MyClass {
    public int publicAttribute;
    protected int protectedAttribute;
    private int privateAttribute;
    int defaultAttribute;

    public void publicMethod() {
        // Code
    }

    protected void protectedMethod() {
        // Code
    }

    private void privateMethod() {
        // Code
    }

    void defaultMethod() {
        // Code
    }
}
```

These access modifiers help in encapsulating and controlling the access to the members of a class, ensuring data hiding and security in Java programs.

Methods

Method Signatures

A method signature consists of the method name and the parameter list (type and order of parameters). The return type may also be considered part of the method signature, but it's not required for method overloading.

```
public void methodName(int parameter1, String parameter2) {
    // Method body
}
```

Passing Arguments

- **Passing by Value**: Primitive data types are passed by value, meaning a copy of the value is passed to the method. Changes to the parameter inside the method do not affect the original value.

  ```
  public void modifyValue(int x) {
      x = x + 1; // Changes made to x are local to this method
  }
  ```

- **Passing by Reference**: Objects are passed by reference, meaning the reference to the object is passed to the method. Changes to the object's state inside the method affect the original object.

```
public void modifyObjectValue(MyObject obj) {
    obj.setValue(10); // Changes made to the object's state affect the
original object
}
```

Returning Values

Methods can return values using the `return` statement. The return type of the method must match the type specified in the method signature.

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

`static` Keyword

The `static` keyword is used to create class-level variables and methods. These belong to the class rather than to individual objects of the class. They can be accessed without creating an instance of the class.

- **Static Variables**:

  ```
  public class MyClass {
      static int count;
  }
  ```

- **Static Methods**:

  ```
  public class MyClass {
      static void printMessage() {
          System.out.println("Hello, world!");
      }
  }
  ```

Static methods can be accessed using the class name:

```
MyClass.printMessage();
```

Static variables and methods are shared among all instances of the class and can be accessed directly from the class itself.

These concepts help in organizing code, improving code reusability, and managing resources effectively in Java programs.

Constructors

Default Constructors

A default constructor is a constructor with no parameters. If no constructor is explicitly defined in a class, Java provides a default constructor automatically. It initializes object attributes to their default values.

```
public class MyClass {
    // Default constructor
    public MyClass() {
        // Constructor body
    }
}
```

Parameterized Constructors

Parameterized constructors allow initialization of object attributes with specific values passed as arguments during object creation.

```
public class MyClass {
    int value;

    // Parameterized constructor
    public MyClass(int v) {
        value = v;
    }
}
```

Copy Constructors, Passing Object as a Parameter

A copy constructor creates a new object by copying the attributes of an existing object. It takes an object of the same class as a parameter.

```
public class MyClass {
    int value;

    // Copy constructor
    public MyClass(MyClass obj) {
        value = obj.value;
    }
}
```

Constructor Overloading

Constructor overloading allows a class to have multiple constructors with different parameter lists. Java differentiates between constructors based on the number and types of parameters.

```
public class MyClass {
    int value;

    // Default constructor
    public MyClass() {
        value = 0;
    }

    // Parameterized constructor
    public MyClass(int v) {
        value = v;
    }
```

```java
    // Overloaded constructor
    public MyClass(int v1, int v2) {
        value = v1 + v2;
    }
}
```

In the example above, `MyClass` has three constructors: a default constructor, a parameterized constructor with one parameter, and an overloaded constructor with two parameters.

Constructors are essential for initializing objects and setting up their initial state. They provide flexibility in object creation and initialization in Java.

Strings

`String` Class

In Java, the `String` class represents a sequence of characters. Strings in Java are immutable, meaning once created, their values cannot be changed.

```java
String str = "Hello, World!";
```

Common `String` Methods

- **charAt(int index)**: Returns the character at the specified index.

  ```java
  char ch = str.charAt(0); // Returns 'H'
  ```

- **contains(CharSequence s)**: Checks if the string contains the specified sequence of characters.

  ```java
  boolean contains = str.contains("World"); // Returns true
  ```

- **format(String format, Object... args)**: Returns a formatted string using the specified format string and arguments.

  ```java
  String formattedString = String.format("Hello, %s!", "John"); // Returns
  "Hello, John!"
  ```

- **length()**: Returns the length of the string.

  ```java
  int length = str.length(); // Returns 13
  ```

- **split(String regex)**: Splits the string into an array of substrings based on the specified regular expression.

  ```java
  String[] parts = str.split(", "); // Splits the string into parts separated
  by ", "
  ```

- **substring(int beginIndex)**: Returns a substring starting from the specified index.

  ```java
  String substring = str.substring(7); // Returns "World!"
  ```

- **substring(int beginIndex, int endIndex)**: Returns a substring from the specified begin index (inclusive) to the specified end index (exclusive).

  ```java
  String substring = str.substring(7, 12); // Returns "World"
  ```

- **toLowerCase()**: Converts all characters in the string to lowercase.

  ```java
  String lowercase = str.toLowerCase(); // Returns "hello, world!"
  ```

- **toUpperCase()**: Converts all characters in the string to uppercase.

  ```java
  String uppercase = str.toUpperCase(); // Returns "HELLO, WORLD!"
  ```

- **trim()**: Removes leading and trailing whitespace from the string.

  ```java
  String trimmed = "  Hello, World!  ".trim(); // Returns "Hello, World!"
  ```

These are some of the commonly used methods provided by the `String` class in Java for manipulating and working with strings. They enable various operations such as substring extraction, case conversion, searching, and splitting.

User Input (Scanner Class)

In Java, the `Scanner` class is commonly used to read user input from the console. It provides various methods to read different types of input, such as integers, floating-point numbers, and strings.

Using `Scanner` Class for User Input

1. **Import `Scanner` class**: First, import the `Scanner` class from the `java.util` package.

   ```java
   import java.util.Scanner;
   ```

2. **Create a Scanner object**: Create an instance of the `Scanner` class to read input.

   ```java
   Scanner scanner = new Scanner(System.in);
   ```

3. **Read input**: Use the `Scanner` object's methods to read input from the console.

   ```java
   System.out.println("Enter your name: ");
   String name = scanner.nextLine(); // Read a line of text
   ```

   ```java
   System.out.println("Enter your age: ");
   int age = scanner.nextInt(); // Read an integer
   ```

4. **Close the Scanner**: It's good practice to close the `Scanner` object after reading input to release system resources.

   ```java
   scanner.close();
   ```

Command-line Arguments

Java programs can also accept command-line arguments, which are passed to the `main` method when the program is executed from the command line.

```java
public class Main {
    public static void main(String[] args) {
        // args is an array of strings containing command-line arguments
        // Process command-line arguments here
    }
}
```

Command-line arguments can be accessed from the `args` array within the `main` method. Each element of the array corresponds to a command-line argument passed to the program.

```java
public class Main {
    public static void main(String[] args) {
        // Display all command-line arguments
        for (int i = 0; i < args.length; i++) {
            System.out.println("Argument " + (i + 1) + ": " + args[i]);
        }
    }
}
```

Command-line arguments are useful for passing information to a Java program when it is executed, such as configuration settings or file paths. They can be accessed and processed as needed within the program.

Unit 3: Inheritance, Packages, and Interfaces

Inheritance

Inheritance is a key concept in object-oriented programming (OOP) that allows a class to inherit properties and behavior from another class. It promotes code reuse and establishes a hierarchical relationship between classes.

Basics of Inheritance

- **Base Class (Superclass)**: The class whose properties and behavior are inherited by another class is called the base class or superclass.
- **Derived Class (Subclass)**: The class that inherits properties and behavior from another class is called the derived class or subclass.
- **Syntax**: In Java, inheritance is achieved using the `extends` keyword.

```
// Base class
class Vehicle {
    // Properties and methods
}

// Derived class inheriting from Vehicle
class Car extends Vehicle {
    // Additional properties and methods
}
```

Types of Inheritance

1. **Single Inheritance**: A subclass inherits from only one superclass.
2. **Multiple Inheritance**: A subclass inherits from multiple superclasses. Java does not support multiple inheritance for classes, but it supports it for interfaces through interface implementation.
3. **Multilevel Inheritance**: A subclass inherits from another subclass, forming a chain of inheritance.
4. **Hierarchical Inheritance**: Multiple subclasses inherit from the same superclass.
5. **Hybrid Inheritance**: It combines multiple types of inheritance, such as single, multiple, or multilevel inheritance.

`extends` Keyword

The `extends` keyword is used to establish an inheritance relationship between classes in Java.

```
class Subclass extends Superclass {
    // Subclass definition
}
```

`super` Keyword

The `super` keyword is used to refer to the superclass or call superclass constructors and methods from the subclass.

- **Referring to Superclass Members**: Use `super` to access superclass members (fields and methods) from the subclass.

  ```
  class Subclass extends Superclass {
      void display() {
          super.display(); // Call superclass method
          // Additional subclass code
      }
  }
  ```

- **Calling Superclass Constructor**: Use `super()` to call the superclass constructor from the subclass constructor.

```java
class Subclass extends Superclass {
    Subclass() {
        super(); // Call superclass constructor
        // Subclass constructor code
    }
}
```

In summary, inheritance allows classes to inherit properties and behavior from other classes, promoting code reuse and establishing a hierarchical relationship between classes. Java supports various types of inheritance, and the `extends` and `super` keywords are used to implement and work with inheritance in Java programs.

Polymorphism: Method Overloading, Overriding & Dynamic Dispatch

Method Overloading

Method overloading allows a class to have multiple methods with the same name but different parameter lists. The methods must have different parameter types or a different number of parameters.

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

    double add(double a, double b) {
        return a + b;
    }
}
```

Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method signature (name and parameters) must be the same.

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

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

Overriding Object Class Methods

Java provides a set of methods in the `Object` class that can be overridden in subclasses to provide custom behavior. Commonly overridden methods include:

- **equals(Object obj)**: Compares two objects for equality.
- **toString()**: Returns a string representation of the object.
- **finalize()**: Called by the garbage collector before reclaiming the object's memory.
- **hashCode()**: Returns a hash code value for the object.

```java
class Student {
    int id;
    String name;

    // Overriding equals method
    @Override
    public boolean equals(Object obj) {
        // Custom equality check logic
    }

    // Overriding toString method
    @Override
    public String toString() {
        return "Student[id=" + id + ", name=" + name + "]";
    }
}
```

Method Dynamic Dispatch

Method dynamic dispatch (or dynamic method dispatch) is the process by which the correct version of a method is invoked at runtime, based on the actual type of the object.

```java
Animal animal = new Dog();
animal.makeSound(); // Dynamic dispatch invokes Dog's makeSound() method
```

In the example above, even though the reference `animal` is of type `Animal`, the `makeSound()` method of the `Dog` class is invoked because `animal` is referring to a `Dog` object. This allows for polymorphic behavior, where the same method call can exhibit different behavior depending on the actual type of the object at runtime.

Polymorphism, achieved through method overloading, overriding, and dynamic dispatch, allows for flexible and reusable code by enabling objects of different types to be treated uniformly.

Interfaces

Defining and Implementing Interfaces

Interfaces in Java define a contract that classes can implement, specifying methods that implementing classes must provide. They are used to achieve abstraction and provide a way to achieve multiple inheritance in Java through interface implementation.

Defining Interfaces

An interface is declared using the `interface` keyword followed by the interface name and a list of method signatures (without method bodies).

```java
interface Shape {
    double area();
    double perimeter();
}
```

Implementing Interfaces

To implement an interface, a class uses the `implements` keyword followed by the interface name. The class must provide implementations for all the methods declared in the interface.

```java
class Circle implements Shape {
    double radius;

    // Implementing area method
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

    // Implementing perimeter method
    @Override
    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}
```

Multiple Inheritance Using Interfaces

Java supports multiple inheritance through interfaces, as a class can implement multiple interfaces. This allows a class to inherit from multiple sources, providing flexibility in code design.

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

interface Colorable {
    void setColor(String color);
}

class Rectangle implements Drawable, Colorable {
    // Implementing draw method
    @Override
    public void draw() {
        // Draw rectangle
    }

    // Implementing setColor method
    @Override
    public void setColor(String color) {
        // Set rectangle color
    }
}
```

In the example above, the `Rectangle` class implements both the `Drawable` and `Colorable` interfaces, allowing it to provide implementations for methods defined in both interfaces.

Interfaces provide a way to achieve abstraction, decoupling the definition of methods from their implementation. They also enable code reuse and multiple inheritance, making Java programs more flexible and maintainable.

Abstract Class & Final Class

Abstract Class

An abstract class in Java is a class that cannot be instantiated directly and may contain abstract methods, which are declared but not implemented in the abstract class itself. Abstract classes are used to define a common interface for a group of subclasses while allowing subclasses to provide specific implementations for abstract methods.

Abstract Class Syntax

An abstract class is declared using the `abstract` keyword. It can contain both abstract and non-abstract methods.

```
abstract class Shape {
    abstract double area(); // Abstract method
    double perimeter() {    // Non-abstract method
        return 0;
    }
}
```

Abstract Methods

An abstract method is declared using the `abstract` keyword and does not have an implementation in the abstract class. Subclasses must provide implementations for all abstract methods.

```
abstract double area(); // Abstract method declaration
```

Differences from Interfaces

- Abstract classes can have constructors, member variables, and non-abstract methods, while interfaces cannot.
- A class can extend only one abstract class, but it can implement multiple interfaces.
- Abstract classes are used to provide a common base for a group of related classes, while interfaces are used to define a contract for implementing classes.

Final Class

A final class in Java is a class that cannot be subclassed, meaning no other class can inherit from it. Final classes are typically used when a class should not be extended or when all its methods are already implemented and should not be overridden.

Final Class Syntax

A final class is declared using the `final` keyword.

```
final class FinalClass {
    // Class definition
}
```

Final Method

In addition to final classes, individual methods can also be marked as final. A final method cannot be overridden by subclasses.

```
class Parent {
    final void display() {
        // Method implementation
    }
}

class Child extends Parent {
    // This will cause a compile-time error
    void display() {
        // Method implementation
    }
}
```

Summary

- Abstract classes provide a way to define a common interface for a group of subclasses and allow for both abstract and non-abstract methods.
- Final classes cannot be subclassed, and final methods cannot be overridden.
- Abstract classes are used when a class should not be instantiated directly, while final classes are used when a class should not be extended.

Packages in Java

Packages in Java are used to group related classes, interfaces, and sub-packages, making the code easier to manage and modularize. They help avoid naming conflicts and can also control access to classes and class members (methods and fields) due to their access levels.

Creating Packages

To create a package, you use the `package` keyword at the top of your Java source file. Each file can only declare one package, and all types (classes, interfaces, enums) declared in the file will belong to that package.

```java
// File: MyPackageClass.java
package mypackage;

public class MyPackageClass {
    // Class contents
}
```

To organize files into packages, the directory structure should reflect the package name. For example, a class `MyPackageClass` in the package `mypackage` should be located in a directory named `mypackage`.

Importing Packages

You can use types from other packages by importing them. The `import` statement is used for this purpose and can be placed after the package declaration and before class declarations in a Java file.

- **Importing a Single Class**: Imports a specific class from a package.

```java
import java.util.ArrayList;
```

- **Importing an Entire Package**: Uses the asterisk (*) as a wildcard character to import all classes from the specified package.

```java
import java.util.*;
```

Access Rules: Access Control Within Packages

Java uses access modifiers to control access levels for classes, constructors, methods, and variables. The access levels impact how members can be accessed from within their own package and from other packages.

- **`public`**: The member is accessible from any other class or package.
- **`protected`**: The member is accessible within its own package and by subclasses (including those in other packages).
- **`default` (no modifier)**: The member is accessible only within its own package. If no access modifier is specified, the default access level is applied.
- **`private`**: The member is accessible only within its own class.

Example: Access Control

```java
package packageOne;

public class ClassOne {
    public void publicMethod() {} // Accessible from any class
    protected void protectedMethod() {} // Accessible within package and
subclasses
    void defaultMethod() {} // Accessible only within packageOne
    private void privateMethod() {} // Accessible only within ClassOne
}
```

If another class in a different package tries to access these methods, only `publicMethod()` and, under certain conditions, `protectedMethod()` (from a subclass) would be accessible.

Packages and access modifiers together provide a robust mechanism for encapsulating and organizing code, ensuring that internal implementations are well-protected and that the public interface of classes is clearly defined.

Unit 4: Exception Handling and Multithreading

Exception Handling in Java

Exception handling in Java is a powerful mechanism that handles runtime errors to maintain normal application flow. An exception is an event that disrupts the normal flow of the program's instructions.

Errors vs. Exceptions

- **Errors**: Indicate serious problems that a reasonable application should not try to catch. Most errors are abnormal conditions. Examples include `OutOfMemoryError` and `StackOverflowError`.
- **Exceptions**: Are conditions that a reasonable application might want to catch. Exceptions are divided into two categories: checked exceptions (those that must be caught or declared to be thrown) and unchecked exceptions (those that don't need to be explicitly caught or declared thrown).

`try-catch-finally` Blocks

- **try block**: Contains code that might throw an exception.
- **catch block**: Catches and handles the exception thrown by the try block.
- **finally block**: Executes after the try/catch block has completed. The finally block will execute whether or not an exception is caught or thrown. It's typically used for cleanup code.

```
try {
    // Code that may throw an exception
} catch (ExceptionType name) {
    // Code to handle the exception
} finally {
    // Code to be executed after try block ends
}
```

Common Built-in Exceptions

- `NullPointerException`: Attempting to access or modify a null object reference.
- `ArrayIndexOutOfBoundsException`: Accessing an array with an illegal index.
- `ClassCastException`: Attempting to cast an object to a subclass of which it is not an instance.
- `NumberFormatException`: Attempting to convert a string to a numeric type but the string doesn't have an appropriate format.

Throwing Exceptions

- **throw keyword**: Used within a method to throw an exception. Either the method must handle the exception using a try-catch block, or it must be declared to throw the exception using the `throws` keyword in the method signature.

```
public void myMethod() throws MyException {
    throw new MyException("Something went wrong");
}
```

- **throws keyword**: Indicates that a method may throw one or more exceptions. The calling method must handle these exceptions.

Creating Custom Exceptions

You can create custom exceptions by extending the `Exception` class (for checked exceptions) or the `RuntimeException` class (for unchecked exceptions).

```
class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}
```

Custom exceptions allow you to create specific error types for your application, improving readability and maintainability.

Multithreading in Java

Multithreading in Java allows concurrent execution of multiple threads within a single process, enabling better utilization of CPU resources and improved application responsiveness. Here's an overview of key concepts and features:

Concepts of Threads and Processes

- **Process**: A process is an executing instance of a program that has its own memory space, resources, and state.
- **Thread**: A thread is the smallest unit of execution within a process. Threads share the same memory space and resources within a process.

Creating Threads

In Java, threads can be created by either extending the `Thread` class or implementing the `Runnable` interface.

- **Extending Thread class**:

```java
class MyThread extends Thread {
    public void run() {
        // Code to be executed in a separate thread
    }
}

// Creating and starting the thread
MyThread thread = new MyThread();
thread.start();
```

- **Implementing Runnable interface**:

```java
class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed in a separate thread
    }
}

// Creating a thread using the runnable object
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Thread Lifecycle

The lifecycle of a thread in Java consists of several states:

- **New**: The thread is in the new state before it is started.
- **Runnable**: The thread is in the runnable state when it's ready to run, but the scheduler has not selected it to be the running thread.
- **Running**: The thread is in the running state when the processor is actively executing its code.
- **Blocked/Waiting**: The thread is in the blocked/waiting state when it's waiting for a resource or another thread to perform a task.
- **Terminated**: The thread is in the terminated state when it has completed its execution.

Thread Priority

Thread priority is used by the scheduler to determine the order of thread execution. Thread priority values range from 1 to 10, where higher values indicate higher priority.

```java
thread.setPriority(Thread.MAX_PRIORITY); // Set highest priority
thread.setPriority(Thread.MIN_PRIORITY); // Set lowest priority
```

Synchronization

Synchronization in Java is used to control access to shared resources by multiple threads. It prevents concurrent access to shared resources, avoiding data corruption and inconsistency.

- **Synchronized methods**:

```java
public synchronized void synchronizedMethod() {
    // Synchronized method body
}
```

- **Synchronized blocks**:

  ```
  synchronized (obj) {
      // Synchronized block
  }
  ```

Thread Exception Handling

Exception handling in threads is similar to exception handling in other Java programs. You can catch exceptions within the `run()` method or propagate them to the caller using `throws` clause.

```
class MyThread extends Thread {
    public void run() {
        try {
            // Code that may throw an exception
        } catch (Exception e) {
            // Handle the exception
        }
    }
}
```

Summary

Multithreading in Java allows concurrent execution of multiple threads within a single process. It enables better utilization of CPU resources, improves application responsiveness, and supports concurrent programming paradigms. Understanding thread concepts, lifecycle, synchronization, and exception handling is crucial for building robust multithreaded applications.

Unit 5: File Handling and Collections Framework

File Handling in Java

File handling in Java involves reading from and writing to files. This can be achieved using streams and various stream classes provided by the `java.io` package.

Streams and Stream Classes

- **Stream**: A sequence of data elements made available over time. In Java, streams are used to perform input and output operations.
- **Stream Classes**: Java provides a variety of stream classes for handling input and output operations. These include byte streams (`InputStream`, `OutputStream`) and character streams (`Reader`, `Writer`).

`FileInputStream` and `FileOutputStream`

- **`FileInputStream`**: Used for reading data from a file as a stream of bytes.
- **`FileOutputStream`**: Used for writing data to a file as a stream of bytes.

```
// Example of using FileInputStream to read from a file
try (FileInputStream fis = new FileInputStream("input.txt")) {
    int data;
    while ((data = fis.read()) != -1) {
```

```
        // Process the data
    }
} catch (IOException e) {
    e.printStackTrace();
}

// Example of using FileOutputStream to write to a file
try (FileOutputStream fos = new FileOutputStream("output.txt")) {
    String data = "Hello, world!";
    fos.write(data.getBytes());
} catch (IOException e) {
    e.printStackTrace();
}
```

Creation, Reading, and Writing Files

- **Creating a File**: You can create a new file using the `File` class.

```
File file = new File("newfile.txt");
boolean created = file.createNewFile();
```

- **Reading from a File**: You can use file input streams (`FileInputStream`, `FileReader`) to read from a file.
- **Writing to a File**: You can use file output streams (`FileOutputStream`, `FileWriter`) to write to a file.

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
    writer.write("Hello, world!");
} catch (IOException e) {
    e.printStackTrace();
}
```

Closing Streams

It's important to close streams after using them to release system resources.

```
try (FileInputStream fis = new FileInputStream("input.txt")) {
    // Code to read from the input stream
} catch (IOException e) {
    e.printStackTrace();
} // Stream will be closed automatically after the try block
```

Summary

File handling in Java involves reading from and writing to files using streams and stream classes. `FileInputStream` and `FileOutputStream` are used for byte-level file handling, while `FileReader` and `FileWriter` are used for character-level file handling. It's essential to properly handle exceptions and close streams after using them to avoid resource leaks.

Collections Framework in Java

The Collections Framework in Java provides a unified architecture for representing and manipulating collections of objects. It includes interfaces, implementations, and algorithms for working with collections efficiently.

Overview and Hierarchy

The Collections Framework includes several key interfaces and classes organized in a hierarchy:

- **Interfaces**: `Collection`, `List`, `Set`, `Map`, etc.
- **Classes**: `ArrayList`, `LinkedList`, `HashSet`, `HashMap`, etc.
- **Hierarchy**:
    - `Collection`
        - `List`: Ordered collection with duplicates allowed (e.g., `ArrayList`, `LinkedList`)
        - `Set`: Unordered collection with no duplicates (e.g., `HashSet`)
    - `Map`: Key-value pairs (e.g., `HashMap`)

`ArrayList`

- Implements the `List` interface.
- Resizable-array implementation of the `List` interface.
- Provides dynamic resizing, fast random access, and fast iteration.
- Efficient for accessing elements by index, but less efficient for insertion and deletion in the middle of the list.

```java
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("C++");
```

`LinkedList`

- Implements the `List` interface.
- Doubly-linked list implementation of the `List` interface.
- Provides fast insertion and deletion operations at both ends of the list.
- Less efficient for random access compared to `ArrayList`.

```java
LinkedList<Integer> list = new LinkedList<>();
list.add(1);
list.add(2);
list.add(3);
```

`HashMap`

- Implements the `Map` interface.
- Hash table-based implementation of the `Map` interface.
- Stores key-value pairs.
- Provides constant-time performance for the basic operations (get and put) on average.

```
HashMap<String, Integer> map = new HashMap<>();
map.put("Java", 1);
map.put("Python", 2);
map.put("C++", 3);
```

`HashSet`

- Implements the `Set` interface.
- Hash table-based implementation of the `Set` interface.
- Stores unique elements, does not allow duplicates.
- Provides constant-time performance for the basic operations (add, remove, contains) on average.

```
HashSet<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Orange");
```

The For-Each Loop

The for-each loop, also known as the enhanced for loop, provides a simple way to iterate over collections and arrays in Java.

```
ArrayList<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
list.add("C++");

for (String language : list) {
    System.out.println(language);
}
```

The for-each loop iterates over each element in the collection sequentially, without the need for explicit indexing or iterators.

The Collections Framework in Java provides a powerful and efficient way to work with collections of objects. Understanding its interfaces and implementations, such as `ArrayList`, `LinkedList`, `HashMap`, and `HashSet`, along with the for-each loop, is essential for effective Java programming.