Object Oriented Thinking and Java Basics

Need for OOP Paradigm

1. **Improvement Over Procedural Programming**:
    - **Issues in Procedural Programming**:
        - Procedural programming (e.g. C, COBOL, PASCAL, FORTRAN) follows a top-down approach.
        - Programs are divided into functions based on tasks.
        - Data and functions are separate, which can lead to data inconsistency and redundancy.
        - Global variables allow unrestricted data access across functions, making code difficult to maintain and extend.
        - Lack of data hiding and operator overloading features.
2. **Benefits of Object-Oriented Programming**:
    - **Bottom-Up Approach**:
        - OOP follows a bottom-up approach, focusing on smaller problems that are combined into larger solutions.
        - It allows for easier program design and maintenance.
    - **Data Encapsulation**:
        - Data and functions are encapsulated into a single unit known as a class.
        - Data is hidden and cannot be accessed by external functions, ensuring security and data integrity.
    - **Modular Design**:
        - Programs are divided into objects based on real-world entities.
        - Objects communicate with each other through methods and messages.
        - Each object has its own data, leading to better organization and management of code.
    - **Reusability**:
        - Codes can be reused across different platforms and projects, saving time and effort.
        - Inheritance allows classes to inherit properties and methods from other classes.
    - **Maintainability**:
        - OOP makes code easy to modify, extend, and maintain due to its modular design and encapsulation.
        - Changes in one part of the program have minimal impact on other parts.
    - **Operator Overloading**:

    - Allows for overloading of operators, providing flexibility and expressiveness in code.

- Object-oriented programming provides a structured and efficient approach to program design.
- It addresses issues found in procedural programming and offers features like data encapsulation, modular design, reusability, and maintainability.

Summary of OOP Concepts

[Basic Concepts of OOPs]

1. Object

- Objects are the basic runtime entities in object-oriented systems.
- They may represent a person, place, bank account, table of data, or any other item the program handles.
- Objects also represent user-defined data such as structures, time, and lists (e.g. Vector).
- An object contains both the data and functions.

Example:

```
class Student {
    // Data members
    String name;
    int age;
    int marks;

    // Functions
    void getData() {
        // Function implementation
    }

    void display() {
        // Function implementation
    }
}
```

2. Class

- Class is a blueprint of an object.
- It is a collection of similar types of objects.
- Once a class has been defined, objects belonging to that class can be created.

Example:

```
class Human {
    // Data members
    String name;
    int age;

    // Functions
    void speak() {
        // Function implementation
    }
}
```

3. Data Abstraction

- Data abstraction is the process of hiding the internal details of an application from the outside world.
- It emphasizes only sharing relevant functionality while hiding lower-level details.

Types:

- **Abstract Class**: A class containing abstract methods that may or may not have an implementation.
- **Interface**: A set of methods that classes must implement.

4. Encapsulation

- Encapsulation is the wrapping up of data and functions into a single unit.
- It combines data members and member functions.
- Data encapsulation enables data hiding or information hiding.

5. Inheritance

- Inheritance is the process of creating a new class from an existing class.
- The existing class is called the base class or super class, while the new class is called the derived class or sub-class.
- The child class contains all the properties of the parent class and its own additional properties.

6. Polymorphism

- Polymorphism means "many forms."
- It refers to the ability of a function to operate differently based on the inputs or data types.
- A person can have different characteristics at the same time (e.g., employee, father, husband, and son).

Types:

- **Compile-Time Polymorphism (Ad Hoc or Static Polymorphism)**: Method overloading and operator overloading.
- **Run-Time Polymorphism (Pure or Dynamic Polymorphism)**: Method overriding.

7. Overloading

- Overloading occurs when two or more methods in the same class have the same name but different parameters.
- Types of overloading include:
  - **Operator Overloading**
  - **Function Overloading**

8. Dynamic Binding

- Binding is the process of connecting one program to another.
- Dynamic binding means the code associated with a procedure call is known only at the time of execution.

9. Message Passing

- Message passing is a process or type of communication between processes.
- Communication is done by sending messages from a sender to a receiver through a network.

Example:

```
class Animal {
    void speak() {
        System.out.println("Animal is speaking");
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("Dog is barking");
    }
}

// In the main method, the method speak() will exhibit dynamic binding:
public static void main(String[] args) {
    Animal animal = new Dog();
    animal.speak(); // Output: Dog is barking
}
```

Coping with Complexity

Coping with complexity in object-oriented programming (OOP) involves managing and simplifying complex software systems by breaking them down into smaller, more manageable pieces. OOP provides several mechanisms to achieve coping with complexity:

Mechanisms:

1. Abstraction

- Abstraction involves creating and defining abstract classes and interfaces to establish a common structure for related objects.
- It allows developers to focus on the essential features of an object while hiding unnecessary details.

2. Encapsulation

- Encapsulation controls unauthorized modification and access to data, helping maintain the integrity of the object.
- It prevents external code from directly accessing an object's internal state, promoting data hiding.

3. Inheritance

- Inheritance reduces redundancy and simplifies code reuse by allowing the extension of existing classes.
- Derived classes can inherit attributes and methods from base classes, reducing the need to reimplement common functionality.

4. Polymorphism

- Polymorphism allows methods to operate on various types of objects, providing flexibility in code.
- Methods can be overridden in derived classes to provide specific implementations for different types.

5. Modularity

- Modularity involves breaking the system into smaller classes or modules.
- Each module focuses on a specific task or functionality, reducing overall complexity and improving maintainability.

6. Design Patterns

- Design patterns help manage complexity by promoting best practices and structuring code effectively.
- They offer reusable solutions to common programming problems, improving code quality and efficiency.

7. Composition

- Composition can be used to build objects that consist of other objects, each responsible for a specific aspect of functionality.
- It enables the creation of complex objects through the combination of simpler components.

Abstraction Mechanisms

Abstraction is a process that hides the implementation details from the user. It provides a clear interface while concealing the underlying complexity.

Advantages

1. **Security Enhancement**: By hiding the internal workings, abstraction enhances the security of the program.

Ways to Achieve Abstraction Mechanism

There are two ways to achieve abstraction in object-oriented programming:

1. Abstract Class

- An abstract class contains the `abstract` keyword in its declaration.
- Objects cannot be created directly from an abstract class.
- An abstract class may or may not contain abstract methods.

Abstract Methods

- An abstract method contains the `abstract` modifier at the time of declaration.
- It does not contain a method body and ends with a semicolon.
- Abstract methods are typically used when an action is common, but implementations may differ.

```
// Example of an abstract class
abstract class Fruits {
    // Abstract method
    abstract public void taste();
}
```

2. Interface

- Interfaces contain only abstract methods.
- In Java, interfaces are implemented using the `implement` keyword.
- Methods in an interface are by default `public` and `abstract`.
- Variables in an interface are by default `public`, `static`, and `final`.

Example:

```
// Abstract class
abstract class Bike {
    abstract void run();
}

// Subclass extending the abstract class
class Honda extends Bike {
    void run() {
        System.out.println("Running safely");
    }

    // Main method to test the code
    public static void main(String[] args) {
        Bike obj = new Honda();
        obj.run(); // Calls the run method from the Honda class
    }
}
```

Notes on Classes and Objects

- Classes are blueprints for creating objects.
- Objects are instances of a class and can have attributes and behaviors.

In the example provided, the `Bike` class is an abstract class with an abstract method `run()`. The `Honda` class extends `Bike` and provides an implementation for the `run()` method.

History of Java

1. Java was developed by James Gosling and his team at Sun Microsystems in the early 1990s.
2. It was officially released in 1995 with the launch of Java 1.0, offering platform independence, object-oriented design, and built-in support for networked applications.
3. Key milestones in the history of Java include releases such as Java 1.1, Java 5 (Tiger), Java 8, Java 9, Java 10, Java 11, and the latest LTS release, Java 17.
4. Java became widely used for various applications, including web, mobile, enterprise, and desktop applications, due to its robustness, platform independence, and large ecosystem of libraries and frameworks.

Java is a programming language created by James Gosling from Sun Microsystems in 1991. It is a class-based, object-oriented, simple, and high-level general-purpose language.

Java Categories

Java is divided into three main categories:

1. **J2SE - Java 2 Standard Edition**
    - Used for client applications.
2. **J2EE - Java 2 Enterprise Edition**
    - Used for developing server-side applications.
3. **J2ME - Java 2 Micro Edition**
    - Used for developing mobile and wireless applications.

Java Terminology

1. **Java Compiler**
    - Converts source code into byte code that can be executed by the JVM (Java Virtual Machine).
2. **JVM (Java Virtual Machine)**
    - A virtual machine that executes Java byte code.
    - Provides an environment for running Java programs and translates byte code into machine code.
3. **JDK (Java Development Kit)**
    - A software development environment used for developing Java applications.
    - Includes a set of tools, libraries, and compiler.
4. **JRE (Java Runtime Environment)**
    - A subset of JDK, including JVM and libraries.
    - Does not contain development tools.

Java Buzzwords

Java is known for its features that make it a versatile and powerful programming language.

1. Simple

- Java is easy to learn and has a simple syntax, making it easy to understand.
- It has a rich set of APIs (Application Protocol Interface).
- Java has removed complicated features such as pointers.
- It includes automatic garbage collection to manage memory efficiently.

2. Object-Oriented

- Java supports object-oriented concepts such as objects, classes, data abstraction, encapsulation, inheritance, and polymorphism.

3. Platform Independent

- Java compiler converts source code into byte code, making it platform independent.
- Byte code can be executed on multiple platforms such as Windows, Linux, macOS, and Ubuntu.
- Java's slogan "Write Once, Run Anywhere" (WORA) reflects this capability.

4. Secure

- Java is more secure compared to other languages such as C and C++.
- Java programs run inside a JVM, providing a layer of security.
- Code is compiled to byte code, which is not human-readable.

5. Robust

- Java uses strong memory management.
- It supports exception handling and type checking mechanisms.

6. Architecture Neutral

- Java is architecture neutral.
- The size of primitive data types is fixed across all architectures.
- For example, in Java, the `int` data type occupies 4 bytes of memory, regardless of the architecture.

7. Portable

- Java supports platform-independent and architecture-neutral features, making it portable across different systems.

8. Dynamic

- Java is a dynamic language that supports dynamic memory allocation.
- It reduces memory wastage and improves application performance.
- Memory space can be dynamically allocated using the `new` operator.

9. Interpreter

- Java supports both compiler and interpreter.
- It is considered a high-level interpreter programming language.

10. High Performance

- Java uses JIT (Just-In-Time) compiler to improve performance.
- The JIT compiler is enabled by default.

11. Multi-Threading

- Java supports multi-threading, allowing multiple programs to execute concurrently and simultaneously.
- A thread is a lightweight process and a single unit of execution.
- Multi-threading shares a common memory space and doesn't occupy memory for each thread.

12. Distributed

- Java enables users to create distributed applications.
- Technologies such as RMI (Remote Method Invocation) and Enterprise JavaBeans are used for creating distributed applications.

Data Types and Variables

1. Primitive Data Types

- Predefined data types in Java.
- Variables' size and type are specified.
- Additional methods are available for these types.

Numeric Data Types

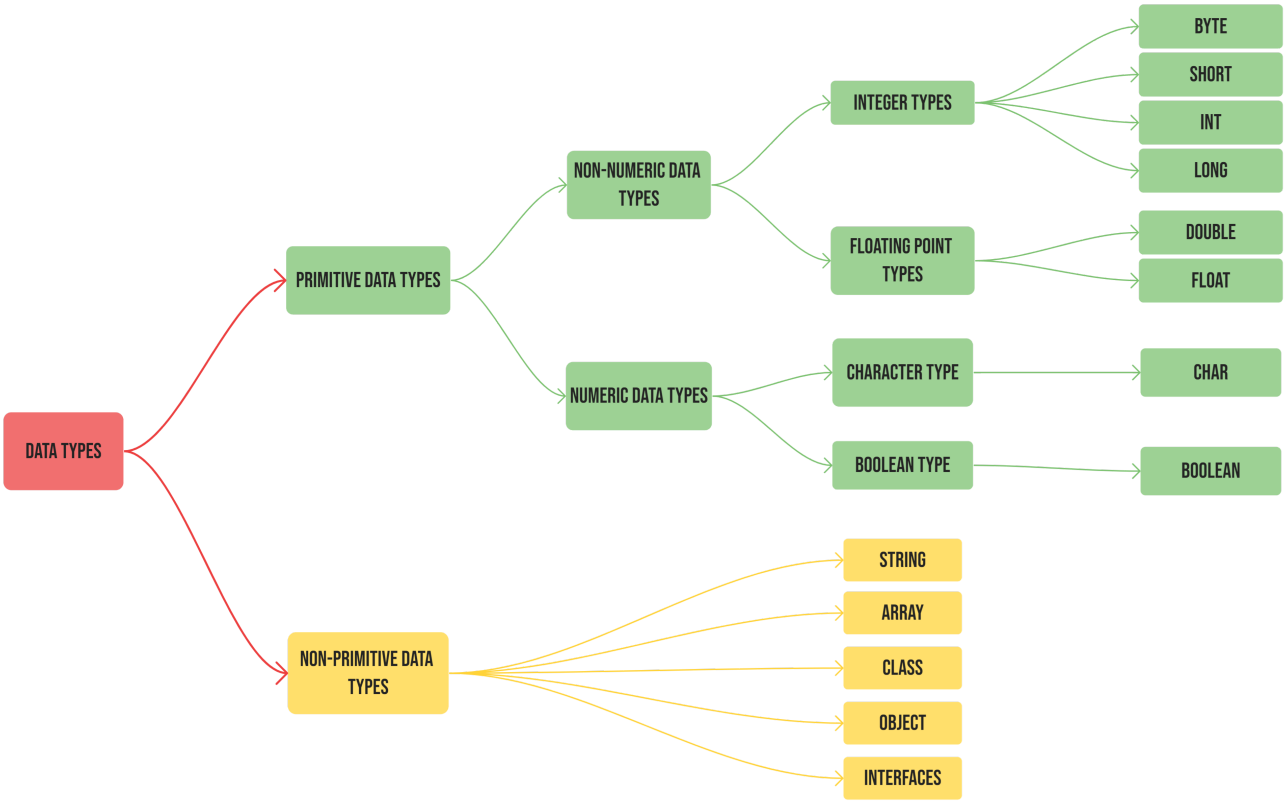
- **Integer Types**
  - `byte`
  - `short`
  - `int`
  - `long`
- **Floating Point Types**
  - `float`
  - `double`

Non-Numeric Data Types

- **Character Type**
  - `char`
- **Boolean Type**

- `boolean`

Flowchart for Data Types and Its Subtypes



Description of Data Types

Data Type	Bytes	Description
byte	1	Integer values from -128 to 127
short	2	Integer values from -32768 to 32767
int	4	Integer values from -2147483648 to 2147483647
long	8	Integer values from -9223372036854775808 to 9223372036854775807
float	4	Single-precision floating-point numbers
double	8	Double-precision floating-point numbers
boolean	1	Boolean values (true or false)
char	2	Unicode character (single character)

2. Non-Primitive Data Types

- Created by programmers, also known as reference variables or object references.
- Variables reference the memory location where data is stored.
- Examples include arrays, classes, and interfaces.

3. Variables in Java

- Variables are named memory locations used to store data values.
- Variables can be containers that hold data values.

Types of Variables

- **Local Variables**
  - Declared inside the body of a method.
  - Access modifiers cannot be used.
  - Should be declared and initialized before the first use.
  - No default value.
- **Instance Variables**
  - Declared inside the class but outside the body of a method.
  - Created when an object is created using the `new` keyword.
  - Have default values (e.g., zero for numeric types).
- **Class Variables (Static Variables)**
  - Declared with the static keyword in a class but outside a method.
  - Stored in static memory.
  - Created when the program starts and destroyed when the program stops.

Dynamic Initialization of Variables

- Initialization can be performed during runtime, known as dynamic initialization of variables.
- Example:

```
double pi = 3.14; // Compile-time initialization
double area = pi * 1.5; // Runtime initialization
```

Scope and Lifetime of Variables

Scope of a Variable

- The **scope** of a variable refers to the area or section of a program where the variable can be accessed.
- Variables declared within a method (local variables) have scope within that method.
- Variables declared outside of a method but within a class have a broader scope and can be accessed from methods within the class.
- Examples:

```
class DemoScope {
    public static void main(String[] args) {
        int x = 10; // Scope is within the main method

        if (x == 10) {
            int y = 20; // Scope is within the 'if' block
            System.out.println("x and y: " + x + " " + y);
        }

        x = x + 2;
        // System.out.println(y); // Compile-time error: Cannot access 'y'
outside of 'if' block
        System.out.println("x is " + x);
    }
}
```

- In the example, the variable `y` is declared inside the `if` block, so its scope is limited to that block. It cannot be accessed outside the block.

Lifetime of a Variable

- The **lifetime** of a variable refers to how long the variable stays alive in memory.
- For local variables, the lifetime is the duration of the method execution.
- For instance variables, the lifetime is the duration of the object's life.
- Examples:

```
class DemoLifetime {
    public static void main(String[] args) {
        for (int x = 0; x < 3; x++) {
            int y = 1; // Lifetime of 'y' is the loop iteration
            System.out.println("y is " + y);
            y = 100; // Compile-time error: cannot modify 'y' in the next
iteration
            System.out.println("y now is " + y);
        }
    }
}
```

- In the example, the local variable `y` is declared within the loop block, so its lifetime is limited to each iteration of the loop.

Arrays and Operators

Arrays

- An array is a linear data structure.
- Arrays are used to store multiple values in a single variable.

- Arrays in Java are non-primitive data types that store elements of a similar data type in memory.
- Arrays in Java can store both primitive and non-primitive types of data.
- Individual values in an array are called elements, and the elements are stored in contiguous memory locations.

Types of Arrays

1. ### 1-Dimensional Array
    - Uses a single index to store elements.
    - Syntax: `datatype[] arrayName = new datatype[size];`
    - Example:

```
int a[] = {33, 3, 4, 53};
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
// Output: 33, 3, 4, 53
```

2. ### Multi-Dimensional Array (Array of Arrays)
    - An array of many arrays in a single variable.
    - Allows storage of multiple arrays.
    - Syntax: `datatype[][] arrayName = new datatype[x][y];`
    - Example:

```
int a[][] = { {21, 22, 23}, {54, 54, 14} };
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        System.out.print(a[i][j] + " ");
    }
    System.out.println();
}
// Output:
// 21 22 23
// 54 54 14
```

Addition of Two Matrices

- You can perform addition of two matrices using arrays.
- Example:

```
class ArrayAdd {
    public static void main(String args[]) {
        int A[][] = {
            {1, 2, 3},
            {2, 4, 5},
```

```
            {4, 4, 5}
        };
        int B[][] = {
            {11, 2, 3},
            {13, 15, 4},
            {23, 25, 4}
        };

        int C[][] = new int[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                C[i][j] = A[i][j] + B[i][j];
                System.out.print(C[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

- In this example, two 3x3 matrices are defined (`A` and `B`), and their sum is stored in matrix `C`. The resulting matrix `C` is printed.

Expressions and Control Statements

1. Expressions:

Expressions are combinations of variables, operators, and method calls that evaluate to a single value. They can be used to perform calculations or make decisions in Java programs.

- An expression is a collection of operators and operands that represent a specific value.
- **Operator**: A symbol that performs tasks such as arithmetic, logical, and conditional operations.
- **Operand**: Values on which the operator performs the task.

Types of Expressions

1. **Infix Expression**
    - The operator is used between the operands.
    - Example: `a + b`
2. **Prefix Expression**
    - The operator is used before the operand.
    - Example: `+a b`
3. **Postfix Expression**
    - The operator is used after the operand.
    - Example: `a b+`
4. **Control Statements:** Control statements are used to control the flow of execution in a Java program. They include decision-making statements like if, else if, and else, as well

as looping statements like for, while, and do-while. Switch statements are also used for multi-way branching based on a value.

Here is the information formatted in markdown.md format with topics, subtopics, and points:

2. Control Statements

- Control statements are used to control the flow of execution based on conditions.
- Types of control statements include:
  - Branching statements
  - Looping statements
  - Jumping statements

Branching Statements

- **Conditional**: Also known as decision-making statements.
  - `if` statement
  - `if-else` statement
  - `if-else-if` statement
  - Nested `if` statements
- **Switch** statement with a `default` case.

If Statement

- Syntax:

```
if (condition) {
    // statement;
}
```

- If the condition is true, the statement or block of statements inside the `if` block is executed.

Example

```
public class IfStatementExample {
    public static void main(String[] args) {
        int num = 70;
        if (num < 100) {
            System.out.println("Number is less than 100");
        }
    }
}
```

If-Else Statement

- Syntax:

```
if (condition) {
    // statement;
} else {
    // else statement;
}
```

- If the condition is true, the statement inside the `if` block is executed, otherwise the statement in the `else` block is executed.

Example

```
public class IfElseExample {
    public static void main(String[] args) {
        int num = 20;
        if (num < 50) {
            System.out.println("Number is less than 50");
        } else {
            System.out.println("Number is greater than or equal to 50");
        }
    }
}
```

If-Else-If Statement

- Syntax:

```
if (condition1) {
    // statement1;
} else if (condition2) {
    // statement2;
} else if (condition3) {
    // statement3;
} else {
    // default statement;
}
```

- Used to check multiple conditions. If one of the conditions is true, the corresponding statement is executed. If none of the conditions are true, the final `else` block acts as a default statement.

Example

```
public class IfElseIfExample {
    public static void main(String[] args) {
        int num = 12043;
        if (num < 100 && num ≥ 10) {
            System.out.println("It's a 2-digit number");
```

```
        } else if (num < 1000 && num ≥ 100) {
            System.out.println("It's a 3-digit number");
        } else if (num < 10000 && num ≥ 1000) {
            System.out.println("It's a 4-digit number");
        } else if (num < 100000 && num ≥ 10000) {
            System.out.println("It's a 5-digit number");
        } else {
            System.out.println("Number is not between 1 & 99999");
        }
    }
}
```

Nested If Statements

- Syntax:

```
if (condition1) {
    // code for condition1;
    if (condition2) {
        // code for condition2;
    }
}
```

- Used to evaluate multiple conditions in a nested manner. If condition1 is true, the code for condition1 executes, then condition2 is evaluated and its corresponding code executes.

Example

```
public class NestedIfExample {
    public static void main(String[] args) {
        int num = 70;
        if (num < 100) {
            System.out.println("Number is less than 100");
            if (num > 50) {
                System.out.println("Number is greater than 50");
            }
        }
    }
}
```

Here is the information formatted in markdown.md format with topics, subtopics, and points:

Switch-Case Statement

Syntax

```
switch (expression) {
    case value1:
        // statement;
```

```
        break;
    case value2:
        // statement;
        break;
    case valueN:
        // statement;
        break;
    default:
        // default statement;
}
```

Flowchart

```
Start
↓
Switch (condition/expression)
↓
Case value1:
    True → Statement 1; break;
↓
Case value2:
    True → Statement 2; break;
↓
Case value3:
    True → Statement 3; break;
↓
Default:
    → Default statement;
↓
Exit
```

Example

```
public class WeekDays {
    public static void main(String[] args) {
        int day = 2;
        switch (day) {
            case 1:
                System.out.println("Monday");
                break;
            case 2:
                System.out.println("Tuesday");
                break;
            case 3:
                System.out.println("Wednesday");
                break;
            case 4:
                System.out.println("Thursday");
                break;
```

```
            case 5:
                System.out.println("Friday");
                break;
            case 6:
                System.out.println("Saturday");
                break;
            case 7:
                System.out.println("Sunday");
                break;
            default:
                System.out.println("Invalid");
                break;
        }
    }
}
```

- The switch-case statement is used when there are multiple choices or options, and different actions need to be performed based on each choice.
- The switch-case statement works with primitive data types and enumerated types.
- The `default` statement is optional and serves as a fallback when no other case matches the provided expression.
- The `break` statement is used inside the switch to terminate the current case statement sequence.

Here's the information formatted in markdown.md format with topics, subtopics, and points:

Looping Statements (Iteration Statements)

Iteration statements allow a program's execution to repeat one or more statements. The three main types of loops are `for`, `while`, and `do-while`.

For Loop

- Loops are used to execute a set of statements repeatedly until a particular condition is satisfied.
- The `for` loop combines initialization, condition, and increment or decrement in one line.

Syntax

```
for (initialization; condition; increment/decrement) {
    // body of the loop
}
```

Flowchart

```
Initialization
↓
```

```
Condition checking
↓
True → Statement → Increment/Decrement → Condition checking
↓
False
```

Example

```
class ExamForLoop {
    public static void main(String[] args) {
        System.out.println("The values from 1 to 10 are: ");
        for (int i = 1; i ≤ 10; i++) {
            System.out.println(i);
        }
    }
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

While Loop

- Repeats a statement or block while a particular condition is true.
- The number of iterations in a `while` loop is not fixed.

Syntax

```
while (boolean condition) {
    // body of the loop
}
```

Flowchart

```
Condition checking
↓
True → Statement → Condition checking
↓
False
```

Example

```
class ExamWhileLoop {
    public static void main(String[] args) {
        int i = 10;
```

```
        while (i > 1) {
            System.out.println(i);
            i--;
        }
    }
}
```

Output:

```
10 9 8 7 6 5 4 3 2
```

Do-While Loop

- The `do-while` loop condition is evaluated after the execution of the loop body.

Syntax

```
do {
    // statement;
} while (condition);
```

Flowchart

```
Statement
↓
Condition checking
↓
True → Statement
↓
False
```

Example

```
class ExamDoWhile {
    public static void main(String[] args) {
        int i = 10;
        do {
            System.out.println(i);
            i--;
        } while (i > 1);
    }
}
```

Output:

```
10 9 8 7 6 5 4 3 2
```

Jump Statements

Jump statements transfer control to another part of the program. There are two types: `break` and `continue`.

Break Statement

- The `break` statement is used to terminate the loop.
- It can be used inside a loop.

Flowchart

```
Start
↓
Condition checking
↓
True → Statement → Loop body
↓
False → Break from loop
```

Example

```
class BreakExample {
    public static void main(String[] args) {
        int month = 5;
        switch (month) {
            case 1:
                System.out.println("January");
                break;
            case 2:
                System.out.println("February");
                break;
            case 3:
                System.out.println("March");
                break;
            case 4:
                System.out.println("April");
                break;
            case 5:
                System.out.println("May");
                break;
            case 6:
                System.out.println("June");
                break;
            case 7:
                System.out.println("July");
                break;
```

```
            case 8:
                System.out.println("August");
                break;
            case 9:
                System.out.println("September");
                break;
            case 10:
                System.out.println("October");
                break;
            case 11:
                System.out.println("November");
                break;
            case 12:
                System.out.println("December");
                break;
            default:
                System.out.println("Invalid month");
                break;
        }
    }
}
```

Continue Statement

- The `continue` statement is used inside loops.
- When encountered inside a loop, control directly jumps to the beginning of the loop for the next iteration, skipping the execution of the current iteration's remaining statements.

Example

```
public class ContinueExample {
    public static void main(String[] args) {
        for (int j = 0; j ≤ 6; j++) {
            if (j == 4) {
                continue;
            }
            System.out.println(j);
        }
    }
}
```

Output:

```
0 1 2 3 5 6
```

Type Conversion and Casting

Implicit (Automatic/Widening) Conversion

- Also known as widening type conversion.

- The process of converting a value from one datatype to another datatype automatically.
- Occurs when assigning a smaller datatype value to a larger datatype variable.
- No automatic conversion is supported from numeric types to `char` or `boolean`.

Example

```
class TypeConversion {
    public static void main(String args[]) {
        int a = 10;
        long b = a;
        float c = a;
        System.out.println(a + " " + b + " " + c);
        // Output: 10 10 10.0
    }
}
```

Explicit (Manual/Narrowing) Conversion

- Also known as narrowing type conversion or type casting.
- The process of converting higher datatype values to lower datatype values explicitly.
- When assigning a higher datatype value to a lower datatype variable, some data might be lost.
- The casting operator `()` is used.

Example

```
class TypeCasting {
    public static void main(String args[]) {
        float a = 100.5f;
        double b = 75.15;
        int c = (int) b; // Convert double to int
        long d = (long) a; // Convert float to long

        System.out.println(c);
        System.out.println(d);
        // Output: 75
        // Output: 100
    }
}
```

In the examples, you can see how implicit and explicit conversions work in Java. Implicit conversions are automatic and typically involve widening the data type, while explicit conversions are manual and often involve narrowing the data type.

Simple Java Program

```
public class HelloWorld {
    public static void main(String[] args) {
```

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

Explanation:

- `public class HelloWorld`: Defines a class named `HelloWorld`.
- `public static void main(String[] args)`: Defines the `main` method, which is the entry point of the program. It accepts an array of strings as arguments.
- `System.out.println("Hello, world!");`: Prints "Hello, world!" to the console.

Concepts of Classes, Objects, and Constructors

1. **Classes:** In Java, a class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that type will have.
   Here is the information formatted in markdown.md format with topics, subtopics, and points:

Class Fundamentals

Classes are a core aspect of object-oriented programming. They define the structure of objects, including their properties and methods. Classes can contain fields, methods, constructors, blocks, nested classes, and interfaces.

Components of a Class

1. **Fields**
   - Variables that hold data specific to each instance of a class.
2. **Methods**
   - Functions that define the behaviors or actions of a class.
   - Methods operate on the fields of the class and often manipulate data.
3. **Constructors**
   - Special methods used to initialize objects.
   - Constructors have the same name as the class and do not have a return type.
4. **Blocks**
   - Code blocks inside a class, including static blocks, initialization blocks, and instance initializer blocks.
5. **Nested Classes and Interfaces**
   - Classes can contain other classes or interfaces within their scope.

Class Syntax

```
class ClassName {
    // Fields
```

```
    type fieldName;

    // Methods
    void methodName() {
        // method body
    }
}
```

- **Class**: Represents a group of objects with common properties and serves as a template or blueprint.
- **Creating a Class**: You can define a class using the syntax above and include fields and methods.

Creating an Object

2. **Objects:** Objects are instances of classes. They represent real-world entities and encapsulate data (attributes) and behavior (methods).

- To create an object of a class, use the `new` keyword.

Syntax

```
ClassName objectName = new ClassName();
```

Initializing an Object

There are three ways to initialize an object:

1. **By reference variable**:
    - Directly assign values to the object's fields.

    ```
    ClassName object = new ClassName();
    object.fieldName = value;
    ```

2. **By method**:
    - Use a method within the class to initialize fields.

    ```
    class Student {
        int rollNo;
        String name;

        void insertRecord(int r, String n) {
            rollNo = r;
            name = n;
        }

        void displayInfo() {
            System.out.println("Roll No: " + rollNo + ", Name: " + name);
    ```

```
    }
}

public class TestStudent {
    public static void main(String[] args) {
        Student s = new Student();
        s.insertRecord(420, "AA");
        s.displayInfo();
    }
}
```

3. **By constructor**:
    - Use a constructor to initialize the object.

```
class Employee {
    int id;
    String name;
    float salary;

    Employee(int i, String n, float s) {
        id = i;
        name = n;
        salary = s;
    }

    void display() {
        System.out.println(id + " " + name + " " + salary);
    }
}

public class TestEmployee {
    public static void main(String[] args) {
        Employee e1 = new Employee(420, "AA", 200000.0f);
        e1.display();

        Employee e2 = new Employee(421, "PS", 200001.0f);
        e2.display();
    }
}
```

Difference between object and clours?

Aspect	Object	Class
Definition	An object is an instance of a class.	A class is a blueprint or template for creating objects.
Nature	An object is a real-world entity such as a pen, pencil, student, etc.	A class is a group of similar objects and is a logical entity.
Entity	An object is a physical entity.	A class is a logical entity.

Aspect	Object	Class
Creation	Objects are created using the `new` keyword.	Classes are declared using the `class` keyword.
Memory Allocation	Memory is allocated when the `new` keyword is used to create an object.	Classes themselves do not allocate memory; they are templates for creating objects.

Constructors

3. **Constructors:** Constructors are special methods in a class used for initializing objects. They have the same name as the class and are called when an object is created. Constructors can be used to set initial values for object properties.

- **Definition**: A constructor is a special type of method used for automatically initializing an object.
- **Invocation**: Constructors are invoked whenever an object is created using the `new` keyword.
- **Purpose**: Mainly used to initialize the variables of an object.

Rules for Creating Constructors

1. Constructor name must be the same as its class name.
2. No explicit return type.
3. Constructors are defined inside the class.

Syntax

```
class ClassName {
    ClassName() {
        // Constructor body
    }
}
```

Types of Constructors

1. **Default Constructor**
    - No-argument constructor.
    - Automatically inserted by the Java compiler if no other constructors are present.

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

public class Main {
```

```
    public static void main(String[] args) {
        A obj = new A();
    }
}
```

2. **Parameterized Constructor**
    - Constructor with a specific number of parameters.
    - Allows different values to be provided to distinct objects.

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

    Student(int i, String n) {
        id = i;
        name = n;
    }

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(143, "Surya");
        s1.display();

        Student s2 = new Student(225, "Reba");
        s2.display();
    }
}
```

Example: Calculate Area of a Rectangle Using Constructor

```
class Rectangle {
    int length;
    int breadth;

    // Constructor
    Rectangle(int l, int b) {
        length = l;
        breadth = b;
    }

    void calculateArea() {
        int area = length * breadth;
        System.out.println("Area of Rectangle: " + area);
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10, 5);
        rect.calculateArea();
    }
}
```

Methods and Access Control

1. **Methods:** In Java, methods are functions defined within a class. They perform actions and may return a value. Methods can have parameters that allow them to receive input data.
   Here is the information formatted in markdown.md format with topics, subtopics, and points:

A method is a block of statements that performs a specific task. Methods in Java are similar to functions in other programming languages. They define the behavior of an object and must be declared inside a class.

Characteristics

- **Return type**: Specifies the type of value that the method returns.
- **Parameters**: A list of inputs the method accepts.
- **Name**: The name of the method.

Advantages

- **Code Reusability**: Methods enable code to be reused across different parts of an application.
- **Code Optimization**: Methods help in organizing code and optimizing it for readability and maintainability.

Creating a Method

Syntax

```
class ClassName {
    <access specifier> <return type> <method name>(parameters) {
        // block of statements
    }
}
```

Example: Banking System

The following program demonstrates a simple banking system where you can deposit and withdraw amounts from an account. It includes an `Account` class with `deposit()` and `withdraw()` methods.

```
import java.util.Scanner;

class Account {
    private int accountNumber;
    private String name;
    private String accountType;
    private long balance;

    // Scanner for user input
    Scanner sc = new Scanner(System.in);

    // Method to deposit
    public void deposit() {
        long amt;
        System.out.println("Enter the amount to deposit:");
        amt = sc.nextLong();
        balance += amt;
        System.out.println("Balance after deposit: " + balance);
    }

    // Method to withdraw
    public void withdraw() {
        long amt;
        System.out.println("Enter the amount to withdraw:");
        amt = sc.nextLong();

        if (balance ≥ amt) {
            balance -= amt;
            System.out.println("Balance after withdrawal: " + balance);
        } else {
            System.out.println("Your balance is less than the amount.
Transaction failed.");
        }
    }
}
```

This example includes methods for depositing and withdrawing money from an account, as well as handling transactions and updating the account balance.

2. **Access Control:** Java provides four access control modifiers to restrict access to classes, variables, methods, and constructors:
    - `public`: Accessible from any other class.
    - `private`: Accessible only within the same class.
    - `protected`: Accessible within the same package or subclasses.
    - Default (no modifier): Accessible only within the same package.

The "this" Keyword

`this` is a reserved keyword in Java used to invoke constructors, methods, and static members of a class. It serves to eliminate confusion between class attributes and method

parameters with the same name. The primary purpose of the `this` keyword is to refer to the current instance of a class.

Usage of `this` Keyword

1. **Refer to Current Class Instance Variable**
   - `this` can be used to refer to an instance variable of the current object.
2. **Call Current Class Methods**
   - `this` can be used to call methods of the current class.
3. **Call Current Class Constructors**
   - `this` can be used to call a constructor of the current class.
4. **Pass as Argument in Method Call**
   - `this` can be passed as an argument in method calls.
5. **Pass as Argument in Constructor Call**
   - `this` can be passed as an argument in constructor calls.

Example

```
class Customer {
    int cust_id;
    String cust_name;
    String cust_location;

    // Constructor
    Customer(int cust_id, String cust_name, String cust_location) {
        this.cust_id = cust_id;
        this.cust_name = cust_name;
        this.cust_location = cust_location;
    }

    // Method
    void display() {
        System.out.println("Customer ID: " + cust_id);
        System.out.println("Customer Name: " + cust_name);
        System.out.println("Customer Location: " + cust_location);
    }
}

public class Cust {
    public static void main(String[] args) {
        // Create a new customer object
        Customer cust = new Customer(1001, "Raj", "Hyd");
        // Call the display method
        cust.display();
    }
}
```

In this example, the `Customer` class uses the `this` keyword in the constructor to refer to the current instance variables (`cust_id`, `cust_name`, and `cust_location`). In the `main`

method, a `Customer` object is created and the `display` method is called to print the customer's details.

2. **Avoiding Ambiguity:** It's commonly used to disambiguate between instance variables and parameters with the same name in constructors or methods. For example, `this.variableName` refers to the instance variable, while `variableName` refers to the local variable.
3. **Passing Current Object:** The `this` keyword can also be used to pass the current object as a parameter to other methods or constructors.
4. **Chaining Constructors:** In constructors, `this()` can be used to call another constructor in the same class, enabling constructor chaining. This allows one constructor to call another constructor of the same class with different parameters.

Garbage Collection

Garbage collection in Java is the process by which Java programs automatically manage memory. It involves the automatic release of memory used by objects that are no longer needed, thus optimizing the performance of the application.

- Java programs compile to bytecode that can run on the Java Virtual Machine (JVM).
- When a Java program runs on the JVM, objects are created dynamically using the `new` operator.
- Dynamically allocated objects must be manually released using the `delete` operator.
- The garbage collector finds unused objects and deletes them to free up memory.
- Garbage collection in Java happens automatically during the program's runtime.

Finalize() Method

- The `finalize()` method is a protected method of the `java.lang.Object` class.
- It is available to all objects and can be overridden in custom classes.
- This method performs final operations or cleanup tasks on an object before it is removed from memory.
- Once the `finalize()` method completes, the garbage collector destroys the object.
- The garbage collector is a module of the JVM.

Syntax

```
protected void finalize() {
    // Cleanup code
}
```

Example

```
class Garbage {
    public void finalize() {
        System.out.println("Object is garbage collected");
    }

    public static void main(String[] args) {
        Garbage a = new Garbage();
        Garbage b = new Garbage();
        a = null;
        b = null;
        System.gc(); // Manually invoking garbage collection
    }
}
```

In this example, two objects of the `Garbage` class are created and then set to `null`. The `System.gc()` method is called to manually invoke garbage collection, which will cause the `finalize()` method to be executed for the objects being collected.

1. **Automatic Memory Management:** Garbage collection is a feature of Java that automatically manages memory by deallocating objects that are no longer needed.
2. **Identification of Unreachable Objects:** The garbage collector periodically scans memory to identify objects that are no longer referenced or reachable by the program.
3. **Reclamation of Memory:** Once unreachable objects are identified, their memory is reclaimed, freeing up resources for new objects.
4. **Prevention of Memory Leaks:** Garbage collection helps prevent memory leaks by automatically deallocating memory occupied by objects that are no longer in use.
5. **Efficient Memory Usage:** By managing memory automatically, garbage collection ensures efficient memory usage in Java programs without requiring manual intervention from developers.

Overloading Methods and Constructors

Method Overloading

Method overloading in Java occurs when a class has multiple methods with the same name but different parameters (different argument types or different numbers of arguments).

1. **Multiple Methods/Constructors with the Same Name:** Overloading allows defining multiple methods or constructors in a class with the same name but different parameter lists.
2. **Different Parameter Lists:** Overloaded methods or constructors must have different parameter lists, either in terms of the number or type of parameters.
3. **Improved Flexibility:** Overloading provides flexibility in method invocation, allowing developers to use the same method name for different operations based on the input provided.

4. **Code Reusability:** Overloading promotes code reusability by allowing multiple methods to share the same name, reducing redundancy and improving code readability.
5. **Compile-Time Resolution:** The appropriate method or constructor to be invoked is determined at compile time based on the number and types of arguments passed.

```
public class OverloadingExample {
    public static void main(String[] args) {
        OverloadingExample example = new OverloadingExample();
        example.print(5);
        example.print("Hello");
    }

    // Method overloading
    public void print(int num) {
        System.out.println("Printing integer: " + num);
    }

    // Overloaded method
    public void print(String message) {
        System.out.println("Printing string: " + message);
    }
}
```

Example

```
class Test {
    public void eat() {
        System.out.println("Hi");
    }

    public void eat(int a) {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.eat(); // Outputs: Hi
        t.eat(5); // Outputs: Hello
    }
}
```

Calculator Example

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

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

public class Demo {
    public static void main(String[] args) {
        SimpleCalculator obj = new SimpleCalculator();
        System.out.println(obj.add(10, 20)); // Outputs: 30
        System.out.println(obj.add(10, 20, 30)); // Outputs: 60
    }
}
```

Constructor Overloading

Constructor overloading in Java is a technique of having multiple constructors in a class that share the same name but have different parameter lists. It is useful for initializing objects in different ways depending on the given parameters.

- **Initialization Flexibility**: Constructor overloading allows different ways of initializing objects with varying sets of parameters.
- **No Return Type**: Constructors do not have a return type, not even `void`. The constructors are called automatically when an object is created.
- **Automatic Invocation**: Constructors are automatically invoked when an object is created using the `new` keyword.
- **Multiple Constructors**: A class can have multiple constructors with different parameter lists. This enables different ways of initializing an object's state.
- **Code Reusability**: Overloaded constructors can be designed to call each other, which promotes code reusability within the class.
- **Consistency**: Constructor overloading helps in maintaining consistent initialization across the class and avoids potential bugs.

Example

```
class ABCD {
    // Default constructor
    ABCD() {
        System.out.println("Default constructor");
    }

    int num1;
    int num2;
    int result;

    // Parameterized constructor
    ABCD(int a, int b) {
        num1 = a;
        num2 = b;
    }
```

```
    void display() {
        result = num1 + num2;
        System.out.println("Result = " + result);
    }
}

public class ConstOverloadExam {
    public static void main(String[] args) {
        // Creating an object using the default constructor
        ABCD obj1 = new ABCD();

        // Creating an object using the parameterized constructor
        ABCD obj2 = new ABCD(10, 20);

        // Displaying the result for obj2
        obj2.display(); // Outputs: Result = 30
    }
}
```

Differences between a constructor and a method:

Constructor	Method
1. A constructor is used to initialize the state of an object.	1. A method is used to expose the behavior of an object.
2. A constructor does not have a return type.	2. A method must have a return type.
3. A constructor is invoked implicitly.	3. A method is invoked explicitly.
4. A constructor name must be the same as the class name.	4. A method name may or may not be the same as the class name.

Method Binding

Here's the information on method binding formatted in Markdown with topics, subtopics, and points:

Method Binding

Binding is the mechanism of connecting a method call to the method body. There are two types of binding:

- **Static Binding (Early Binding)**
- **Dynamic Binding (Late Binding)**

Aspect	Static Binding	Dynamic Binding
Definition	Occurs at compile time.	Occurs at runtime.

Aspect	Static Binding	Dynamic Binding
Also known as	Early binding.	Late binding.
How it's achieved	Through normal function calls, function overloading, and operator overloading.	Achieved with the use of virtual functions.
Execution	Results in faster execution since function calls are resolved before runtime.	Function calls are resolved at runtime.
Flexibility	Offers less flexibility compared to dynamic binding.	Provides more flexibility as different types of objects can be used at runtime.

Passing Arguments to Methods

There are two ways of passing arguments to methods:

- **Pass by Value**
- **Pass by Reference**

Pass by Value

- **Definition**: In pass by value, a copy of the variables is passed to the method.
- **Scope**: The method has access only to the copy of the variables.
- **Example**:

```
class Swapper {
    int a;
    int b;

    Swapper(int x, int y) {
        a = x;
        b = y;
    }

    void swap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
}

public class Main {
    public static void main(String[] args) {
        Swapper obj = new Swapper(10, 20);
        System.out.println("Before swapping: a=" + obj.a + " b=" +
obj.b);
        obj.swap(obj.a, obj.b);
        System.out.println("After swapping: a=" + obj.a + " b=" +
```

```
obj.b);
    }
}
```

Pass by Reference

- **Definition**: In pass by reference, the reference (address) of the actual parameter is passed to the local parameter in the method definition.
- **Example**:

```
class PassByReference {
    int a;
    int b;

    void setData(int a, int b) {
        this.a = a;
        this.b = b;
    }

    void display(PassByReference m) {
        m.a = m.a + 10;
        m.b = m.b - 5;
    }
}

public class Main {
    public static void main(String[] args) {
        PassByReference ref = new PassByReference();
        ref.setData(50, 100);
        System.out.println("Before function call: a = " + ref.a + " b =
" + ref.b);
        ref.display(ref);
        System.out.println("After function call: a = " + ref.a + " b =
" + ref.b);
    }
}
```

- **Output**:

```
Before function call: a = 50 b = 100
After function call: a = 60 b = 95
```

1. **Binding Method Calls to Method Definitions:** Method binding is the process of associating a method call with the method definition.
2. **Static Binding (Compile-Time Binding):** In static binding, the method to be invoked is determined by the compiler based on the type of the reference variable.
3. **Dynamic Binding (Runtime Binding):** In dynamic binding, the method to be invoked is determined at runtime based on the actual type of the object.

4. **Uses of Static Binding:** Static binding is used for private, final, static methods, and constructors.
5. **Uses of Dynamic Binding:** Dynamic binding is used for overridden methods, allowing for polymorphic behavior in Java programs.

Inheritance and Overriding

Inheritance

1. **Inheriting Properties and Behaviors:** Inheritance allows a subclass to inherit properties and behaviors from its superclass.
2. **Extending Superclasses:** Subclasses can extend a superclass using the `extends` keyword, inheriting its attributes and methods.
3. **Customizing Behavior:** Overriding is the process of providing a new implementation for a method in a subclass that is already defined in its superclass.
4. **Polymorphic Behavior:** Overridden methods enable polymorphic behavior, where objects of different classes can be treated as objects of a common superclass.
5. **Flexibility and Reusability:** Inheritance and overriding promote flexibility and code reusability by allowing subclasses to customize and extend the functionality of their superclass.

```
public class InheritanceExample {
    public static void main(String[] args) {
        Car car = new Car();
        car.drive(); // Calls the overridden method in Car class
    }
}

class Vehicle {
    public void drive() {
        System.out.println("Vehicle is driving");
    }
}

class Car extends Vehicle {
    @Override
    public void drive() {
        System.out.println("Car is driving");
    }
}
```

Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This allows the subclass to define the behavior for a method in a way that is specific to the subclass. Method overriding is characterized by:

- **Same method name**: The method in the subclass has the same name as the method in the superclass.
- **Same parameters**: The method in the subclass has the same parameters (signature) as the method in the superclass.
- **Different class**: The method is defined in a subclass that extends the superclass.

Example:

```
class Test {
    public void eat() {
        System.out.println("Hi");
    }
}

class Second extends Test {
    public void eat() {
        System.out.println("Hello");
    }
}

public class Main {
    public static void main(String[] args) {
        Second s = new Second();
        s.eat(); // Output: Hello

        Test t = new Test();
        t.eat(); // Output: Hi
    }
}
```

- **Polymorphism**: Method overriding supports runtime polymorphism, allowing methods to be invoked based on the actual object's type.
- **Dynamic binding**: Method overriding relies on dynamic binding (late binding), where the method to be called is determined at runtime based on the object's type.
- **Access level**: In method overriding, the subclass method should have the same or broader access level (e.g., public, protected) as the superclass method.
- **Return type**: The return type of the subclass method must be the same as, or a subtype of, the return type of the superclass method.
- **Use with inheritance**: Method overriding occurs in the context of inheritance, where a subclass extends a superclass.
- **Annotations**: The `@Override` annotation can be used in Java to explicitly indicate that a method in a subclass is intended to override a method in the superclass. This helps prevent errors due to typos or mismatches in method signatures.

Example:

```
class Vehicle {
    void run() {
```

```
        System.out.println("Vehicle is running");
    }
}

class Bike extends Vehicle {
    @Override
    void run() {
        System.out.println("Bike is riding");
    }
}

public class Main {
    public static void main(String[] args) {
        Bike bike = new Bike();
        bike.run(); // Output: Bike is riding
    }
}
```

Differences between method overloading and method overriding:

Method Overloading	Method Overriding
1. Method overloading is a compile-time polymorphism.	1. Method overriding is a runtime polymorphism.
2. It occurs within the same class.	2. It is performed in two classes with an inheritance relationship.
3. Method overloading may or may not require inheritance.	3. Method overriding needs inheritance.
4. Static binding is used for overloaded methods.	4. Dynamic binding is used for overridden methods.
5. Private and final methods can be overloaded.	5. Private and final methods cannot be overridden.

Exceptions and Parameter Passing

1. **Exceptions Handling:**
    - Java uses exceptions to handle errors and unexpected situations during program execution.
    - Exceptions are caught and handled using try-catch blocks.
    - The `try` block contains the code that may throw an exception, and the `catch` block handles the exception if it occurs.
    - Example:

```
try {
    // Code that may throw an exception
    int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
```

```
        // Handling the exception
        System.out.println("Error: " + e.getMessage());
    }
```

2. **Parameter Passing:**
    - Java uses pass-by-value for parameter passing, where a copy of the actual parameter's value is passed to the method or constructor.
    - For primitive data types, changes made to the parameter within the method do not affect the original value.
    - For objects, the reference to the object is passed, allowing changes made to the object within the method to affect the original object.
    - Example:

```
public void modifyValue(int num) {
    num = num * 2; // Changes to num do not affect the original
value
}

public void modifyArray(int[] arr) {
    arr[0] = 100; // Changes to arr affect the original array
}
```

Recursion and Nested Classes

Recursion

Recursion in Java is a process where a method calls itself continuously until a base case is reached. This self-referential method is known as a recursive method. It is a powerful technique for solving problems by breaking them down into smaller, more manageable problems.

Syntax:

```
return_type methodName() {
    if (baseCase) {
        // Base case code
    } else {
        // Recursive call
        methodName();
    }
}
```

- **Base case**: Recursion must have a base case, which is a condition that terminates the recursion and prevents an infinite loop.
- **Recursive case**: The method should contain at least one recursive call, which is the self-referential call within the method.

- **Memory**: Recursion can consume a lot of memory due to function call stack growth, so care must be taken to avoid stack overflow.
- **Stack frames**: Each recursive call creates a new stack frame, which contains the method's parameters and local variables.
- **Tail recursion**: In certain scenarios, if the recursive call is the last operation in the method (tail recursion), it can be optimized by the compiler for better performance.
- **Performance**: Recursive methods can be elegant and concise, but may have higher memory and processing overhead compared to iterative approaches.
- **Use cases**: Recursion is useful for solving problems with self-similar structure, such as factorial calculations, Fibonacci sequences, tree and graph traversals, and divide-and-conquer algorithms.

Common use cases:

- **Factorial**: Calculating the factorial of a number using recursion.
- **Fibonacci sequence**: Generating the Fibonacci sequence using recursion.
- **Tree and graph traversal**: Recursion is a natural fit for traversing tree and graph data structures.
- **Divide-and-conquer algorithms**: Algorithms like quicksort and mergesort leverage recursion for breaking down problems into smaller subproblems.

Example:

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

2. **Nested Classes:**
    - Java allows classes to be defined within other classes, known as nested classes.
    - Types of nested classes include static nested classes, non-static nested classes (inner classes), local classes, and anonymous classes.
    - Example:

```
public class Outer {
    private int outerVar;

    // Nested class
    public class Inner {
        public void display() {
            System.out.println("Inner class method");
            System.out.println("Outer variable: " + outerVar);
        }
```

```
    }
}
```

Exploring String Class

1. **Immutable Class:**
   - The String class in Java is immutable, meaning its value cannot be changed once created.
   - Example:

```
String str = "Hello";
str.concat(" World"); // This creates a new String object
System.out.println(str); // Output: Hello
```

2. **Common Methods:**
   - The String class provides methods for various string manipulations such as length, substring, indexOf, etc.
   - Example:

```
String str = "Hello, World";
System.out.println(str.length()); // Output: 12
System.out.println(str.substring(7)); // Output: World
```

3. **String Pool:**
   - Java maintains a string pool, a pool of unique String literals, to optimize memory usage.
   - Example:

```
String str1 = "Hello";
String str2 = "Hello";
System.out.println(str1 == str2); // Output: true
```

4. **String Comparison:**
   - String comparison should be done using the equals() method, not the == operator.
   - Example:

```
String str1 = "Hello";
String str2 = new String("Hello");
System.out.println(str1.equals(str2)); // Output: true
```