1. Introduction to Java Programming Language

1.1. 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.

1.1.1. Brief History & Evolution of Java

- **Inception (1991):** Sun Microsystems initiated the Java project under James Gosling, aiming for embedded devices. Java was originally called "Oak".
- **Public Debut (1995):** Java was unveiled, focusing on web applets and its "Write Once, Run Anywhere" (WORA) philosophy.
- **Growth & Refinement:** Subsequent releases (Java 2 and beyond) introduced major platforms (J2SE, J2EE, J2ME), significant language improvements, and vast libraries.
- **Oracle Acquisition (2010):** Oracle took ownership, driving Java's evolution.
- **Modern Era:** Java remains a powerhouse, adapting to cloud computing, big data, and modern development paradigms.

1.1.2. 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.

1.1.3. 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.
- **Big Data:** Tools within the Java ecosystem (like Hadoop, Spark) are widely used for processing vast datasets.
- **Embedded Systems:** Java finds use in certain embedded systems and IoT (Internet of Things) devices.
- **Scientific Applications:** Popular for computation, modeling, and simulation.

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.

1.2. Java Environment Setup & Basic Java Syntax

1.2.1. 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)**: A subset of the JDK, focused on running Java programs. JRE includes JVM along with libraries and other components required to run Java applications but does not include development tools.
- **JDK (Java Development Kit)**: The essential package for developing Java applications. JDK is a full-featured software development kit that includes JRE, compilers, debuggers, and other tools necessary for developing Java applications.

1.2.2. 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.

1.2.3. Structure of a Java Program

A basic Java program consists of:

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

1.2.3.1. Class Declaration

Every Java program begins with a class declaration. The class name should match the filename.

1.2.3.2. Main Method

The main method is the entry point of a Java program. It has the following syntax:

- 'Public' means the class/method is accessible from anywhere.
- 'static' allows the JVM to call this method without creating an object of the class.
- 'void' means the method doesn't return a value.
- 'main' is a special method name.

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

1.2.3.3. Output in Java

Output in Java is typically achieved using the `System.out.println()` method. `System` is a built-in Java class that contains useful members, such as `out`, which is short for "output".

1.2.3.3.1. The `println()` Method

The `println()` method, short for "print line", is used to print a value to the screen (or a file). You should also note that each code statement must end with a semicolon (`;`).

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

1.2.3.3.2. The `print()` Method

There is also a `print()` method, which is similar to `println()`. The only difference is that it does not insert a new line at the end of the output:

```
System.out.print("Hello World! ");
System.out.print("I will print on the same line.");
```

You can also use the `println()` method to print numbers. However, unlike text, we don't put numbers inside double quotes:

```
System.out.println(3);
System.out.println(358);
System.out.println(50000);
System.out.println(3 + 3);
System.out.println(2 * 5);
```

1.2.3.4. Comments

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

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

1.2.3.4.1. Single-line Comments

Single-line comments start with two forward slashes (`//`). Any text between `//` and the end of the line is ignored by Java (will not be executed). This example uses a single-line comment before a line of code:

```java
// This is a comment
System.out.println("Hello World");
```

This example uses a single-line comment at the end of a line of code:

```java
System.out.println("Hello World"); // This is a comment
```

1.2.3.4.2. Multi-line Comments

Multi-line comments start with `/*` and ends with `*/`. Any text between `/*` and `*/` will be ignored by Java. This example uses a multi-line comment (a comment block) to explain the code:

```java
/* The code below will print the words Hello World
to the screen, and it is amazing */
System.out.println("Hello World");
```

1.2.4. 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
   ```

1.2.5. 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.

1.3. Data Types

A variable in Java must be a specified data type:

```
int myNum = 5;               // Integer (whole number)
float myFloatNum = 5.99f;    // Floating point number
char myLetter = 'D';         // Character
boolean myBool = true;       // Boolean
String myText = "Hello";     // String
```

Data types are divided into two groups:

- Primitive data types - includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`
- Non-primitive data types - such as `String`, Arrays and Classes.

1.3.1. Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods. There are eight primitive data types in Java.

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

- **Numeric:**
  - **Integer Types:**

        - `byte` **(8 bits)**: The `byte` data type can store whole numbers from -128 to 127. This can be used instead of `int` or other integer types to save memory when you are certain that the value will be within -128 and 127
        - `short` **(16 bits):**The `short` data type can store whole numbers from -32768 to 32767:
        - `int` **(32 bits):** The `int` data type can store whole numbers from -2147483648 to 2147483647. In general, and in our topic, the `int` data type is the preferred data type when we create variables with a numeric value.
        - `long` **(64 bits):**The `long` data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":
    - **Floating-Point Types:** You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515. The `float` and `double` data types can store fractional numbers. Note that you should end the value with an "f" for floats and "d" for doubles:
        - `float` **(32-bit single precision):**
        - `double` **(64-bit double precision):**
- **Character:**
    - `char` **(16-bit Unicode character):** The `char` data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':
- **Boolean:**
    - `boolean` **(true or false):** Very often in programming, you will need a data type that can only have one of two values, like: YES / NO, ON / OFF, TRUE / FALSE. For this, Java has a `boolean` data type, which can only take the values `true` or `false`

1.3.2. Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects. The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for `String`).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be `null`.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.

Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc.

1.3.3. Type Conversion and Casting

1.3.3.1. Implicit Conversion (Widening)

Java automatically converts smaller data types to larger ones to prevent loss of data. For example, `int` can be implicitly converted to `long`.

`byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);      // Outputs 9
    System.out.println(myDouble);   // Outputs 9.0
  }
}
```

1.3.3.2. Explicit Conversion (Narrowing)

When converting larger data types to smaller ones, explicit casting is required to avoid loss of data. For example: `int myInt = (int) 3.14;`

`double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

1.4. 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.

1.4.1. Naming Rules & Conventions

1.4.1.1. Naming Rules

- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter, and cannot contain whitespace
- Names can also begin with $ and _ (but we will not use it here)
- Names are case-sensitive ("myVar" and "myvar" are different variables)
- Reserved words (like Java keywords, such as `int` or `boolean`) cannot be used as names

1.4.1.2. 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`).

1.4.2. Variables

- **Variable Declaration**: Variables are containers for storing data values. 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
  ```

- **Declare Many Variables:** To declare more than one variable of the **same type**, you can use a comma-separated list:

  ```
  int x = 5, y = 6, z = 50;
  System.out.println(x + y + z);
  ```

- **One Value to Multiple Variables**: You can also assign the **same value** to multiple variables in one line:

  ```
  int x, y, z;
  x = y = z = 50;
  System.out.println(x + y + z);
  ```

1.4.3. Constants (`final` Keyword)

If you don't want others (or yourself) to overwrite existing values, use the `final` keyword (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

- **Declaration:** Constants in Java are declared using the `final` keyword.

  ```
  final int myNum = 15;
  myNum = 20;  // will generate an error: cannot assign a value to a final variable
  ```

- **Immutable:** The value of a constant cannot be changed once initialized.
- By convention, constant names are written in uppercase letters with underscores separating words.

1.4.4. 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.

```
public class Main {
  public static void main(String[] args) {
    // Code here CANNOT use x
    { // This is a block
      // Code here CANNOT use x
      int x = 100;
      // Code here CAN use x
      System.out.println(x);
    } // The block ends here
  // Code here CANNOT use x
  }
}
```

- **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.

1.5. Arrays

An array is a data structure that stores a fixed-size collection of elements of the same data type. Each element is accessed by its index (position) within the array.

1.5.1. 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
  ```

- **Key points**
  - Array indices start at 0 and go up to the length of the array minus 1.
  - Trying to access an element outside the array bounds will result in an `ArrayIndexOutOfBoundsException`.

1.5.2. Multidimensional Arrays

- **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]; // Instantiation 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:

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

Things to remember

- Multidimensional arrays can have more than two dimensions.
- Rows and columns in a multidimensional array can have different lengths.
- Two-dimensional arrays can represent matrices, tables, grids, etc., and are useful for storing and processing structured data in Java.

1.6. Operators

Operators are used to perform operations on variables and values. In the example below, we use the `+` **operator** to add together two values:

```
int x = 100 + 50;
```

Although the `+` operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

```
int sum1 = 100 + 50;        // 150 (100 + 50)
int sum2 = sum1 + 250;      // 400 (150 + 250)
int sum3 = sum2 + sum2;     // 800 (400 + 400)
```

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators

- Logical operators
- Bitwise operators

1.6.1. Arithmetic Operators

Arithmetic operators are used to perform mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

```
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)
```

1.6.2. Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3

Operator	Example	Same As
<<=	x <<= 3	x = x << 3

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

1.6.3. Relational (Comparison) Operators

Relational operators are used to establish relationships between two values. This is important in programming, because it helps us to find answers and make decisions. The return value of a comparison is either `true` or `false`. These values are known as *Boolean values*, and you will learn more about them in the Booleans and If..Else topic.

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

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

1.6.4. Logical Operators

You can also test for `true` or `false` values with logical operators. Logical operators are used to determine the logic between variables or values.

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

```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)
```

1.6.5. Bitwise Operators

Bitwise operators perform bitwise operations on integer operands.

```java
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
```

1.6.6. 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.

- This is also called as a short-hand if else.
- It is known as the **ternary operator** because it consists of three operands.
- It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:

Syntax: `\*variable\* = (\*condition\*) ? \*expressionTrue\* :  \*expressionFalse\*;`

Instead of writing:

```java
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
```

You can simply write:

```java
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

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

1.6.7. Operator Precedence

Java follows a specific order for evaluating expressions with multiple operators (similar to mathematical order of operations). You can find a detailed precedence table online.

```
int x = 5 + 3 * 2;  // x will be 11 (Multiplication first)
boolean isGreater = 10 >= 5; // isGreater will be true
int y = 10;
y++; // Postfix increment, y is now 11
++y; // Prefix increment, y is now 12
int result = (2 > 3) ? 10 : 20; // result will be 20
```

1.7. 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.

1.7.1. Selection Statements

Java has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

1.7.1.1. The if Statement

Use the `if` statement to specify a block of Java code to be executed if a condition is `true`.

```
//  Syntax
if (condition) {
  // block of code to be executed if the condition is true
}
```

Note that `if` is in lowercase letters. Uppercase letters (If or IF) will generate an error.

In the example below, we test two values to find out if 20 is greater than 18. If the condition is `true`, print some text:

```
// Example
if (20 > 18) {
  System.out.println("20 is greater than 18");
}
```

We can also test variables:

```
int x = 20;
int y = 18;
if (x > y) {
  System.out.println("x is greater than y");
}
```

In the example above we use two variables, **x** and **y**, to test whether x is greater than y (using the `>` operator). As x is 20, and y is 18, and we know that 20 is greater than 18, we print to the screen that "x is greater than y".

1.7.1.2. The if-else Statement

Use the `else` statement to specify a block of code to be executed if the condition is `false`.

```
// Syntax
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

```
// Example
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

In the example above, time (20) is greater than 18, so the condition is `false`. Because of this, we move on to the `else` condition and print to the screen "Good evening". If the time was less than 18, the program would print "Good day".

1.7.1.3. The if-else-if Ladder

Use the `else if` statement to specify a new condition if the first condition is `false`.

```
// Syntax
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is
true
} else {
  // block of code to be executed if the condition1 is false and condition2 is
false
}
```

```java
// Example
int time = 22;
if (time < 10) {
  System.out.println("Good morning.");
} else if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

In the example above, time (22) is greater than 10, so the **first condition** is `false`. The next condition, in the `else if` statement, is also `false`, so we move on to the `else` condition since **condition1** and **condition2** is both `false` - and print to the screen "Good evening".

However, if the time was 14, our program would print "Good day."

1.7.1.4. Switch-Case Statements

Instead of writing **many** `if..else` statements, you can use the `switch` statement.

The `switch` statement selects one of many code blocks to be executed:

```java
// Syntax
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The `switch` expression is evaluated once.
- The value of the expression is compared with the values of each `case`.
- If there is a match, the associated block of code is executed.
- The `break` and `default` keywords are optional, and will be described later here

The example below uses the weekday number to calculate the weekday name:

```java
// Example
int day = 4;
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;
}
// Outputs "Thursday" (day 4)
```

1.7.1.4.1. `break` Keyword

When Java reaches a `break` keyword, it breaks out of the switch block.

1.7.1.4.2. `default` Keyword

The `default` keyword specifies some code to run if there is no case match:

```java
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the Weekend");
}
// Outputs "Looking forward to the Weekend"
```

Note that if the `default` statement is used as the last statement in a switch block, it does not need a break.

1.7.2. Looping Statements

Loops can execute a block of code as long as a specified condition is reached. Loops are handy because they save time, reduce errors, and they make code more readable.

1.7.2.1. While Loop

The `while` loop loops through a block of code as long as a specified condition is `true`:

```
// Syntax
while (condition) {
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

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

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

1.7.2.2. Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
// Syntax
do {
  // code block to be executed
}
while (condition);
```

The example below uses a `do/while` loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

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

Do not forget to increase the variable used in the condition, otherwise the loop will never end!

1.7.2.3. For Loop

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:

```
// Syntax
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block. **Statement 2** defines the condition for executing the code block. **Statement 3** is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

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

Statement 1 sets a variable before the loop starts (int i = 0). Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end. Statement 3 increases a value (i++) each time the code block in the loop has been executed.

This example will only print even values between 0 and 10:

```java
for (int i = 0; i <= 10; i = i + 2) {
  System.out.println(i);
}
```

1.7.2.4. 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.

The following example outputs all elements in the **cars** array, using a "**for-each**" loop:

```java
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

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

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

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

1.7.2.5. Nested Loops

It is also possible to place a loop inside another loop. This is called a **nested loop**. The "inner loop" will be executed one time for each iteration of the "outer loop":

```java
// Outer loop
for (int i = 1; i <= 2; i++) {
  System.out.println("Outer: " + i); // Executes 2 times

  // Inner loop
  for (int j = 1; j <= 3; j++) {
    System.out.println(" Inner: " + j); // Executes 6 times (2 * 3)
  }
}
```

1.7.3. Jump Statements

1.7.3.1. `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);
}
```

1.7.3.2. `continue` Statement

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

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

1.7.3.3. `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.

2. Object-Oriented Programming

2.1. Procedure-Oriented vs. Object-Oriented Programming

2.1.1. 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.
6. **Examples:** C, FORTRAN, Pascal, BASIC

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.
6. **Examples:** Java, Python, C++, C#

2.1.2. Differences

Characteristic	Procedure-Oriented	Object-Oriented
Focus	Functions or procedures	Objects (data + behavior)
Program Structure	Top-down approach, functions within a program	Bottom-up approach, objects as building blocks
Data	Global or passed between functions	Encapsulated within objects, accessed mainly via methods
Security	Less secure – data more exposed	Improved security through data hiding and access control
Modularity	Code can be less modular	High modularity due to objects

Characteristic	Procedure-Oriented	Object-Oriented
Reusability	Less reusable	Code reusability enhanced through inheritance and classes
Design Complexity	Suitable for smaller programs	Preferred for large, complex systems due to better modeling of real-world systems

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.

2.2. 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.

2.2.1. 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();
```

2.2.2. Encapsulation

- **Bundling:** Combining data (attributes) and code (methods) that operates on that data within a single unit (class).
- **Protection:** Controlling the visibility of data members using access modifiers (public, private, protected) to protect data integrity and hide implementation details.

Example:

The attributes of a `BankAccount` object are encapsulated within the class, accessible and modifiable mainly through its methods.

```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
    }
}
```

2.2.3. 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.

- **Simplification:** Focusing on essential characteristics and hiding complex details. Exposing only the necessary interface.
- **Levels of Abstraction:** Can be achieved through classes, abstract classes, and interfaces.

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
    }
}
```

2.2.4. 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.

- **Hierarchy:** Creating new classes (subclasses) that inherit properties and behaviors of existing classes (superclasses)
- **Code Reusability:** Subclasses can reuse code from the superclass.

- **Extensibility:** Subclasses can add their own unique properties and behaviors.

Example:

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

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

2.2.5. Polymorphism

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

- **Many Forms:** The ability of an object to take on different forms or behaviours depending on the situation.
- **Method Overloading:** Multiple methods in a class with the same name but different parameters.
- **Method Overriding:** A subclass provides a specific implementation of a method inherited from its superclass.

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.

2.3. Classes and Objects

2.3.1. Creating Classes

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

Syntax

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

Example

```java
public class Car { // 'public' allows access from anywhere
    // Fields (member variables) define attributes
    private String model;  // 'private' limits access to within the class
    private int year;
    private String color;

    // Constructor: Special method to initialize an object
    public Car(String model, int year, String color) {
        this.model = model; // 'this' refers to the current object
        this.year = year;
        this.color = color;
    }

    // Methods define behaviors
    public void startEngine() {
        System.out.println("Engine Starting...");
    }

    public void brake() {
        System.out.println("Braking...");
    }

    // Getters and setters (accessors and mutators) for controlled access
    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    // ... more getters and setters
}
```

2.3.2. Creating Objects

In Java, an object is created from a class. Objects are instances of classes. They are created using the `new` keyword followed by the class constructor

To create an object of `Main`, specify the class name, followed by the object name, and use the keyword `new`: Create an object called "`myObj`" and print the value of x:

```java
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

2.3.2.1. Multiple Objects

You can create multiple objects of one class:

Create two objects of `Main`:

```java
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

2.3.2.2. Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

- Main.java
- Second.java

```java
//Main.java
public class Main {
  int x = 5;
}
```

```java
//Second.java
class Second {
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

2.3.2.3. `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.

```java
public class Person {
    String name;

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

2.4. Class Attributes

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

Create a class called "`Main`" with two attributes: `x` and `y`:

```java
public class Main {
  int x = 5;
  int y = 3;
}
```

Another term for class attributes is **fields**.

2.4.1. Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (`.`):

The following example will create an object of the `Main` class, with the name `myObj`. We use the `x` attribute on the object to print its value:

Create an object called "`myObj`" and print the value of `x`:

```java
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

2.4.2. Modify Attributes

You can also modify attribute values: Set the value of `x` to 40:

```java
public class Main {
  int x;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Or override existing values: Change the value of `x` to 25:

```java
public class Main {
  int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // x is now 25
    System.out.println(myObj.x);
  }
}
```

If you don't want the ability to override existing values, declare the attribute as `final`:

```java
public class Main {
  final int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // will generate an error: cannot assign a value to a final
variable
    System.out.println(myObj.x);
  }
}
```

The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

2.4.3. Attributes of Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other: Change the value of `x` to 25 in `myObj2`, and leave `x` in `myObj1` unchanged:

```java
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    myObj2.x = 25;
    System.out.println(myObj1.x);  // Outputs 5
    System.out.println(myObj2.x);  // Outputs 25
  }
}
```

2.4.4. Multiple Attributes of same Object

You can specify as many attributes as you want:

```java
public class Main {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

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

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

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

2.5. Class Methods

- A **method** is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a method.
- Methods are used to perform certain actions, and they are also known as **functions**.
- Why use methods? To reuse code: define the code once, and use it many times.

2.5.1. Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as `System.out.println()`, but you can also create your own methods to perform certain actions:

Create a method inside Main:

```java
public class Main {
  static void myMethod() {
    // code to be executed
  }
}
```

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later here.
- `void` means that this method does not have a return value. You will learn more about return values later here

2.5.2. Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, `myMethod()` is used to print a text (the action), when it is called:

Inside `main`, call the `myMethod()` method:

```java
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "I just got executed!"
```

A method can also be called multiple times:

```java
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
    myMethod();
    myMethod();
  }
}

// I just got executed!
// I just got executed!
// I just got executed!
```

2.5.3. Method Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `String` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

```java
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }

  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Liam`, `Jenny` and `Anja` are **arguments**.

2.5.3.1. Multiple Parameters

You can have as many parameters as you like:

```java
public class Main {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

2.5.4. Return Values

The `void` keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as `int`, `char`, etc.) instead of `void`, and use the `return` keyword inside the method:

```
public class Main {
  static int myMethod(int x) {
    return 5 + x;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(3));
  }
}
// Outputs 8 (5 + 3)
```

This example returns the sum of a method's **two parameters**:

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(5, 3));
  }
}
// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    int z = myMethod(5, 3);
    System.out.println(z);
  }
}
// Outputs 8 (5 + 3)
```

2.5.5. Access Methods With an Object

Create a Car object named `myCar`. Call the `fullThrottle()` and `speed()` methods on the `myCar` object, and run the program:

```java
// Create a Main class
public class Main {

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Main myCar = new Main();   // Create a myCar object
    myCar.fullThrottle();      // Call the fullThrottle() method
    myCar.speed(200);          // Call the speed() method
  }
}

// The car is going as fast as it can!
// Max speed is: 200
```

2.5.6. 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.

The unique identifier of a method. It consists of:

- **Name:** What the method is called.
- **Parameter List:** The types and order of arguments the method accepts.
- **Return Type:** The type of value returned by the method (`void` if it doesn't return anything).

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

2.5.7. 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
}
```

2.5.8. Returning Values

Methods can return values using the `return` statement.

- The `return` statement exits the method and sends a value back to where the method was called.
- The return type in the method signature must match the data type of the value being returned.
- Methods with a `void` return type don't return anything.

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

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

2.6. Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes

- Have the same name as the class.
- Do not have a return type, not even `void`.

```
// Create a Main class
public class Main {
  int x;  // Create a class attribute
  // Create a class constructor for the Main class
  public Main() {
    x = 5;  // Set the initial value for the class attribute x
  }

  public static void main(String[] args) {
    Main myObj = new Main(); // Create an object of class Main (This will call the
constructor)
    System.out.println(myObj.x); // Print the value of x
  }
}
```

```
// Outputs 5
```

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

2.6.1. Types of Constructors

2.6.1.1. Default Constructors

- If you don't define a constructor, Java provides a no-argument default constructor.
- It typically initializes members to their default values (e.g., 0 for numbers, null for objects).

2.6.1.2. Parameterized Constructors

Parameterized constructors allow initialisation of object attributes with specific values passed as arguments during object creation. Used to provide flexibility when creating objects.

```
public class Student {
    private String name;
    private int rollNumber;

    // Parameterized constructor
    public Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }
}
```

2.6.1.3. Copy Constructors

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 Student {
    // ... (fields and other constructors)

    // Copy constructor
    public Student(Student otherStudent) {
        this.name = otherStudent.name;
        this.rollNumber = otherStudent.rollNumber;
    }
}
```

2.6.2. 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;

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

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

    // 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.

2.7. Modifiers

We divide modifiers into two groups:

- **Access Modifiers** - controls the access level
- **Non-Access Modifiers** - do not control access level, but provides other functionality

2.7.1. Access Modifiers

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

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

For **classes**, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages topic

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
`public`	**Class, Package, Other Packages:** The code is accessible for all classes
`private`	**Class only:** The code is only accessible within the declared class
default	**Class, Package:** The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages topic

Modifier	Description
`protected`	**Class, Package, Subclasses (even in different packages):** The code is accessible in the same package and **subclasses**. You will learn more about subclasses and superclasses in the Inheritance topic

```java
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
    }
}
```

2.7.2. Non-Access Modifiers

For **classes**, you can use either `final` or `abstract`:

Modifier	Description
`final`	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance topic)
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction topics)

For **attributes and methods**, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object

Modifier	Description
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the Inheritance and Abstraction topics
`transient`	Attributes and methods are skipped when serializing the object containing them
`synchronized`	Methods can only be accessed by one thread at a time
`volatile`	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

2.7.2.1. `final`

If you don't want the ability to override existing attribute values, declare attributes as `final`:

```java
public class Main {
  final int x = 10;
  final double PI = 3.14;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 50; // will generate an error: cannot assign a value to a final
variable
    myObj.PI = 25; // will generate an error: cannot assign a value to a final
variable
    System.out.println(myObj.x);
  }
}
```

2.7.2.2. `static`

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.

- **Class-level Methods:** Methods declared `static` don't require an instance of the class to be called. They belong to the class itself. Use Cases:
  - Utility methods not tied to a specific object.
  - The `main` method is `static` since it's your program's entry point.

A `static` method means that it can be accessed without creating an object of the class, unlike `public`.

An example to demonstrate the differences between `static` and `public` methods:

```java
public class Main {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects");
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[ ] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would output an error

    Main myObj = new Main(); // Create an object of Main
    myObj.myPublicMethod(); // Call the public method
  }
}
```

- **Accessing Members:** `static` methods can only directly access other `static` members and cannot use the `this` keyword (since they don't operate on an object).

  ```
  public class MathUtils {
      public static double findCircumference(double radius) {
          return 2 * Math.PI * radius;
      }
  }
  ```

- **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.

2.7.2.3. `abstract`

An `abstract` method belongs to an `abstract` class, and it does not have a body. The body is provided by the subclass:

```
// Code from filename: Main.java
// abstract classabstract class Main {
  public String fname = "John";
  public int age = 24;
  public abstract void study(); // abstract method
}

// Subclass (inherit from Main)
class Student extends Main {
  public int graduationYear = 2018;
  public void study() { // the body of the abstract method is provided here
    System.out.println("Studying all day long");
  }
}
// End code from filename: Main.java

// Code from filename: Second.java
class Second {
  public static void main(String[] args) {
    // create an object of the Student class (which inherits attributes and
methods from Main)
    Student myObj = new Student();

    System.out.println("Name: " + myObj.fname);
    System.out.println("Age: " + myObj.age);
    System.out.println("Graduation Year: " + myObj.graduationYear);
    myObj.study(); // call abstract method  }
}
```

2.8. `String` Class

- In Java, strings are treated as objects of the `String` class. This class provides numerous methods for manipulating and working with strings.
- **Immutability:** It's important to remember that `String` objects in Java are immutable. Once a String is created, its contents cannot be changed.

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

2.8.1. Strings - Special Characters

Because strings must be written within quotes, Java will misunderstand this string, and generate an error:

```
String txt = "We are the so-called "Vikings" from the north.";
```

The solution to avoid this problem, is to use the **backslash escape character**. The backslash (`\`) escape character turns special characters into string characters:

Escape character	Result	Description
'	'	Single quote
"	"	Double quote
\	\	Backslash

The sequence `\"` inserts a double quote in a string:

```
String txt = "We are the so-called \"Vikings\" from the north.";
```

The sequence `\'` inserts a single quote in a string:

```
String txt = "It\'s alright.";
```

The sequence `\\` inserts a single backslash in a string:

```
String txt = "The character \\ is called backslash.";
```

Other common escape sequences that are valid in Java are:

Code	Result
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed

2.8.2. Common `String` Methods

- **String Concatenation:** The `+` operator can be used between strings to combine them. This is called **concatenation**:

  ```
  String firstName = "John";
  String lastName = "Doe";
  System.out.println(firstName + " " + lastName);
  ```

  Note that we have added an empty text (" ") to create a space between firstName and lastName on print. You can also use the `concat()` method to concatenate two strings:

  ```
  String firstName = "John ";
  String lastName = "Doe";
  System.out.println(firstName.concat(lastName));
  ```

- **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.

Additional points

- **String Concatenation:** You can use the `+` operator to join strings together.

- **String Comparison:**
  - Use `.equals()` for content comparison.
  - `==` in the case of strings compares object references, not always the content.
- **StringBuilder:** For frequent modifications, look into the `StringBuilder` class, which is mutable and may be more efficient.

2.9. `Scanner` Class (User Input)

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.

2.9.1. Input Types

In the example above, we used the `nextLine()` method, which is used to read Strings. To read other types, look at the table below:

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

2.9.2. Using `Scanner` Class

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();
```

In the example below, we use different methods to read data of various types:

```java
import java.util.Scanner;

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

    System.out.println("Enter name, age and salary:");

    // String input
    String name = myObj.nextLine();

    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}
```

Note: If you enter wrong input (e.g. text in a numerical input), you will get an exception/error message (like "`InputMismatchException`").

2.10. 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.

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.

- Arguments passed to your program when it's started from the command line.
- Accessed in the `String[] args` parameter of the `main` method.

Example

```
public class CommandLineDemo {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("The first argument is: " + args[0]);
            System.out.println("There were " + args.length + " arguments
passed.");
        } else {
            System.out.println("No command-line arguments provided.");
        }
    }
}
```

Run this from the command line like:

```
java CommandLineDemo hello world
```

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.

3. Inheritance, Packages, and Interfaces

3.1. 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.

3.1.1. 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
  }
  ```

3.1.2. Types of Inheritance

1. **Single Inheritance:** A subclass inherits from only one superclass.

   ```
   class Animal { ... }
   class Dog extends Animal { ... }
   ```

2. **Multiple Inheritance (Not directly supported in Java):** A subclass inheriting from multiple superclasses. Java avoids this using interfaces (we'll cover interfaces later).
3. **Multilevel Inheritance:** A subclass inherits from a class that is itself a subclass.

   ```
   class Animal { ... }
   class Dog extends Animal { ... }
   class GoldenRetriever extends Dog { ... }
   ```

4. **Hierarchical Inheritance:** Multiple subclasses inherit from a single superclass.

   ```
   class Vehicle { ... }
   class Car extends Vehicle { ... }
   class Truck extends Vehicle { ... }
   ```

5. **Hybrid Inheritance:** A combination of multiple inheritance types. This can get complex, and Java doesn't directly support all variations.

3.1.3. `extends` Keyword

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

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

3.1.4. `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.

  ```
  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.

3.1.5. Polymorphism

The word "polymorphism" means "many forms." In Java, it refers to the ability of objects to behave differently depending on their specific type, enabling us to write more flexible and reusable code.

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous topic; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called `Animal` that has a method called `animalSound()`. Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

```java
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}
```

Remember from the Inheritance topic that we use the `extends` keyword to inherit from a class.

Now we can create `Pig` and `Dog` objects and call the `animalSound()` method on both of them:

```java
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class Main {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

3.1.5.1. 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.

- **Definition:** Having multiple methods with the same name within the same class, but with different parameter lists (different number of parameters or different parameter types).
- **Resolution at Compile Time:** The compiler determines at compile time which version of the method to call based on the arguments provided.

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

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

3.1.5.2. 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.

- **Definition:** A subclass redefines a method it inherits from a superclass. The subclass provides its own specific implementation of the inherited method.
- **Resolution at Runtime:** The JVM determines at runtime which version to call (subclass or superclass) based on the type of the object. This is the essence of dynamic dispatch.
- **Use of `@Override` Annotation:** Marking overridden methods with `@Override` helps avoid errors.

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

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

3.1.5.2.1. 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.

```
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 + "]";
    }
}
```

3.1.6. 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.

```
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.

3.2. Interfaces

- An interface is like a contract. It defines a set of methods that a class must implement, ensuring certain behaviors are guaranteed by the class.
- **Abstract:** Interfaces cannot be instantiated directly. They are used to achieve abstraction and provide a way to achieve multiple inheritance in Java through interface implementation.
- **Methods without Bodies:** Methods in an interface are by default abstract (without a body).
- **`implements` Keyword:** Classes implement interfaces using the `implements` keyword.

3.2.1. 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();
}
```

3.2.2. 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;
    }
}
```

3.2.3. 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.

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1. To achieve security - hide certain details and only show the important details of an object (interface).
2. Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Benefits of Interfaces:

- **Polymorphism:** You can treat objects of different classes that implement the same interface uniformly.
- **Multiple Inheritance (via Interfaces):** A class can implement multiple interfaces, overcoming Java's restriction on direct multiple inheritance of classes.
- **Abstraction:** Interfaces help to enforce a separation between interface (what an object can do) and implementation (how it does it).
- **Loose Coupling:** Using interfaces helps to reduce dependencies between classes, making your code more flexible and maintainable.

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.

3.3. Abstraction

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next topic).

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

```java
// Abstract class
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class Main {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

3.3.1. 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.

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

- **`abstract` Keyword:** Abstract classes are declared using the `abstract` keyword.
- **Abstract Methods:** Can contain abstract methods (methods declared without a body, ending with a semicolon). Subclasses **must** implement these methods.
- **Concrete Methods:** Can also have regular methods with implementations.

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

3.3.2. Abstract Method

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.

Example

```java
abstract class Vehicle {
    private String model;

    public Vehicle(String model) {
        this.model = model;
    }

    // Abstract method
    public abstract void startEngine();

    // Concrete method
    public void accelerate() {
        System.out.println("Accelerating...");
    }
}
```

3.3.3. Differences from Interfaces

Feature	Interface	Abstract Class
Instantiation	Cannot be instantiated directly	Cannot be instantiated directly
Method Declaration	Only abstract method declarations	Can have abstract methods AND concrete methods
Implementation	Provides no default implementation	Can provide default implementations for some methods
Multiple Inheritance	A class can implement multiple interfaces	A class can extend only one abstract class

When to Use an Abstract Class

- Common functionality across subclasses, but not all methods make sense at the base level.
- Default implementations exist for some behaviors.
- You want to enforce a certain structure on your class hierarchy.

3.4. Final Class

- **Definition:** A class declared `final` cannot have any subclasses. It's like the end of an inheritance chain.
- Use Cases:
  - Prevent unwanted inheritance.

  - Classes with immutable characteristics (like `String`).
  - Classes with security-sensitive functionality.

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.

3.4.1. Final Class Syntax

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

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

3.4.2. Final Method

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

```java
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.
- You cannot have a class that is both `abstract` and `final`. They represent opposite concepts in terms of inheritance.

3.5. Inner Classes

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

```java
class OuterClass {
  int x = 10;
  class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}

// Outputs 15 (5 + 10)
```

3.5.1. Private Inner Class

Unlike a "regular" class, an inner class can be `private` or `protected`. If you don't want outside objects to access the inner class, declare the class as `private`:

```
class OuterClass {
  int x = 10;

  private class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.y + myOuter.x);
  }
}
```

If you try to access a private inner class from an outside class, an error occurs:

```
Main.java:13: error: OuterClass.InnerClass has private access in OuterClass
OuterClass.InnerClass myInner = myOuter.new InnerClass();       ^
```

3.5.2. Static Inner Class

An inner class can also be `static`, which means that you can access it without creating an object of the outer class:

```
class OuterClass {
  int x = 10;

  static class InnerClass {
    int y = 5;
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.InnerClass myInner = new OuterClass.InnerClass();
    System.out.println(myInner.y);
  }
}

// Outputs 5
```

Note: just like `static` attributes and methods, a `static` inner class does not have access to members of the outer class.

3.5.3. Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```

3.6. Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code.

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.

They provide:

- **Organisation:** Help manage large projects by avoiding naming conflicts.
- **Access Control:** Control the visibility of classes and members.
- **Namespace:** Create a unique namespace for your classes and interfaces.

Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

3.6.1. Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

Java comes with a rich set of built-in packages in the Java API. Examples:

- `java.lang` (String, Math, System, etc.)
- `java.util` (List, ArrayList, Scanner, etc.)
- `java.io` (File, InputStream, etc.)

To use a class or a package from the library, you need to use the `import` keyword:

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

3.6.2. Import a Class

If you find a class you want to use, for example, the `Scanner` class, **which is used to get user input**, write the following code:

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

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Using the `Scanner` class to get user input:

```java
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

3.6.3. Import a Package

There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (`*`). The following example will import ALL the classes in the `java.util` package:

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

3.6.4. User-defined 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.

Package Declaration: At the top of your `.java` files, use the `package` keyword followed by the package name.

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer: root/mypack/MyPackageClass.java

To create a package, use the `package` keyword:

```java
// MyPackageClass.java
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Save the file as **MyPackageClass.java**, and compile it, Then compile the package.

The `-d` keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign "`.`", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

3.6.5. 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.

Access Modifier	Access Within
`public`	Class, Package, Other Packages
`protected`	Class, Package, Subclasses (even in different packages)
`default` (no modifier)	Class, Package
`private`	Class only

- **`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.

3.6.6. 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.

4. Exception Handling and Multithreading

4.1. 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.

4.1.1. 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).

4.1.2. Java try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

```
// Syntax
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Consider the following example:

This will generate an error, because **myNumbers[10]** does not exist.

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }
}
```

The output will be something like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10    at
Main.main(Main.java:4)
```

If an error occurs, we can use `try...catch` to catch the error and execute some code to handle it:

```java
//Example
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

The output will be:

```
Something went wrong.
```

4.1.3. `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.

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

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

```java
// Example
try {
    int result = 10 / 0; // Might throw an ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Error: Cannot divide by zero");
} finally {
    System.out.println("This code always executes.");
}
```

4.2. 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.
- **throws keyword**: Indicates that a method may throw one or more exceptions. The calling method must handle these exceptions.

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

The `throw` statement is used together with an **exception type**. There are many exception types available in Java: `ArithmeticException`, `FileNotFoundException`, `ArrayIndexOutOfBoundsException`, `SecurityException`, etc.

4.2.1. Common Built-in Exceptions

- `ArithmeticException`: Thrown for issues like division by zero.
- `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.
- `IOException`: Signals problems during input/output operations.
- `IllegalArgumentException`: When a method passes an invalid argument.

Throw an exception if **age** is below 18 (print "Access denied"). If age is 18 or older, print "Access granted":

```
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years
old.");
    }
    else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }
}
```

The output will be:

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.    at Main.checkAge(Main.java:4)    at Main.main(Main.java:12)
```

If **age** was 20, you would **not** get an exception:

```
checkAge(20);
```

The output will be:

```
Access granted - You are old enough!
```

4.2.2. 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.

4.2.3. Benefits of Exception Handling

- **Separation of Error-handling Code:** Improves readability and maintainability.
- **Graceful Recovery:** Allows your program to recover from errors instead of crashing.
- **Propagation:** Exceptions can bubble up the call stack if not handled locally.

4.3. Multi-threading in Java

Multi-threading 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:

4.3.1. 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.

4.3.2. Multi-threading Benefits

- **Responsiveness:** UI remains responsive even during long-running tasks.
- **Resource Utilization:** Maximize CPU usage by allowing multiple threads to run concurrently.
- **Simplification:** Can break down complex tasks into smaller, independently running threads.

4.3.3. Creating a Thread

There are two ways to create a thread.

It can be created by extending the `Thread` class and overriding its `run()` method:

4.3.3.1. Extend Syntax

```
public class Main extends Thread {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Another way to create a thread is to implement the `Runnable` interface:

4.3.3.2. Implement Syntax

```
public class Main implements Runnable {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

4.3.4. Running Threads

If the class extends the `Thread` class, the thread can be run by creating an instance of the class and call its `start()` method:

4.3.4.1. Extend Example

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

If the class implements the `Runnable` interface, the thread can be run by passing an instance of the class to a `Thread` object's constructor and then calling the thread's `start()` method:

4.3.4.2. Implement Example

```java
public class Main implements Runnable {
  public static void main(String[] args) {
    Main obj = new Main();
    Thread thread = new Thread(obj);
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

4.3.4.3. Differences between "extending" and "implementing" Threads

The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like: class `MyClass extends OtherClass implements Runnable`.

4.3.5. Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

A code example where the value of the variable **amount** is unpredictable:

```java
public class Main extends Thread {
  public static int amount = 0;

  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println(amount);
    amount++;
    System.out.println(amount);
  }

  public void run() {
    amount++;
  }
}
```

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the `isAlive()` method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

Use `isAlive()` to prevent concurrency problems:

```java
public class Main extends Thread {
  public static int amount = 0;
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    // Wait for the thread to finish
    while(thread.isAlive()) {
    System.out.println("Waiting...");
  }
  // Update amount and print its value
  System.out.println("Main: " + amount);
  amount++;
  System.out.println("Main: " + amount);
  }
  public void run() {
    amount++;
  }
}
```

4.3.6. 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.

4.3.7. Thread Priority

Thread priority is used by the scheduler to determine the order of thread execution.

- Range from 1 (lowest) to 10 (highest), default is 5, where higher values indicate higher priority.
- `thread.setPriority()`, `thread.getPriority()`
- The OS scheduler uses priorities as suggestions, the behavior might be OS-dependent.

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

4.3.8. Thread Exception Handling

Exception handling in threads is similar to exception handling in other Java programs.

- **Uncaught Exceptions:** If an exception isn't caught within a thread's `run` method, it terminates the thread.
- **UncaughtExceptionHandler:** Set a default handler per thread (`thread.setUncaughtExceptionHandler()`) or for all threads (`Thread.setDefaultUncaughtExceptionHandler()`) to log or handle these exceptions.

- You can catch exceptions within the `run()` method or propagate them to the caller using `throws` clause.

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

4.3.9. 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.

- **Critical Sections:** Code blocks that should be executed by only one thread at a time.
- **`synchronized` keyword:** Use on methods or blocks to acquire a lock (monitor) on the object.
- **`wait()`, `notify()`, `notifyAll()`:** For more advanced thread coordination inside synchronized blocks.
- **Synchronized methods**:

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

- **Synchronized blocks**:

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

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.

5. File Handling and Collections Framework

File handling in Java involves reading from and writing to files. Java has several methods for creating, reading, updating, and deleting files.

5.1. File Handling using `File` Class

The `File` class from the `java.io` package, allows us to work with files.

To use the `File` class, create an object of the class, and specify the filename or directory name:

```java
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

The `File` class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
`canRead()`	Boolean	Tests whether the file is readable or not
`canWrite()`	Boolean	Tests whether the file is writable or not
`createNewFile()`	Boolean	Creates an empty file
`delete()`	Boolean	Deletes a file
`exists()`	Boolean	Tests whether the file exists
`getName()`	String	Returns the name of the file
`getAbsolutePath()`	String	Returns the absolute pathname of the file
`length()`	Long	Returns the size of the file in bytes
`list()`	String[]	Returns an array of the files in the directory
`mkdir()`	Boolean	Creates a directory

5.1.1. Create a File

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: `true` if the file was successfully created, and `false` if the file already exists. Note that the method is enclosed in a `try...catch` block. This is necessary because it throws an `IOException` if an error occurs (if the file cannot be created for some reason):

```java
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
File created: filename.txt
```

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "`\`" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

```java
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

5.1.2. Write To a File

In the following example, we use the `FileWriter` class together with its `write()` method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the `close()` method:

```java
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Successfully wrote to the file.
```

5.1.3. Read a File

In the previous topic, you learned how to create and write to a file.

In the following example, we use the `Scanner` class to read the contents of the text file we created in the previous topic:

```java
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files

public class ReadFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

The output will be:

```
Files in Java might be tricky, but it is fun enough!
```

5.1.4. Get File Information

To get more information about a file, use any of the `File` methods:

```java
import java.io.File;  // Import the File class

public class GetFileInfo {   public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

The output will be:

```
File name: filename.txtAbsolute path: C:\Users\MyName\filename.txtWriteable:
trueReadable: trueFile size in bytes: 0
```

Note: There are many available classes in the Java API that can be used to read and write files in Java: `FileReader, BufferedReader, Files, Scanner, FileInputStream, FileWriter, BufferedWriter, FileOutputStream`, etc. Which one to use depends on the Java version you're working with and whether you need to read bytes or characters, and the size of the file/lines etc.

Tip: To delete a file, read our Java Delete Files topic.

5.1.5. Delete a File

To delete a file in Java, use the `delete()` method:

```java
import java.io.File;  // Import the File class

public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

The output will be:

```
Deleted the file: filename.txt
```

5.1.6. Delete a Folder

You can also delete a folder. However, it must be empty:

```java
import java.io.File;

public class DeleteFolder {
  public static void main(String[] args) {
    File myObj = new File("C:\\Users\\MyName\\Test");
    if (myObj.delete()) {
      System.out.println("Deleted the folder: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the folder.");
    }
  }
}
```

The output will be:

```
Deleted the folder: Test
```

5.2. File Handling using `Streams` Class

5.2.1. Streams and Stream Classes

File handling in Java can be achieved using streams and various stream classes provided by the `java.io` package.

- **Stream**: A sequence of data elements made available over time. In Java, streams are used to perform input and output operations.
- Types:
  - **Byte Streams:** Handle raw binary data (files, network).
  - **Character Streams:** Handle character-based data (text files).
- **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`).

5.2.2. `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.

```java
// 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();
}
```

5.2.3. `FileOutputStream` to Write to File

You can use file output streams (`FileOutputStream`, `FileWriter`) to write to a file.

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

```java
import java.io.FileOutputStream;
import java.io.IOException;

public class WriteToFile {
    public static void main(String[] args) {
        try (FileOutputStream outputStream = new
FileOutputStream("myNewFile.txt")) {
            String text = "Hello, this is some text for the file.";
            byte[] data = text.getBytes();
            outputStream.write(data);
            System.out.println("Data written successfully!");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

5.2.4. `FileInputStream` to Read from a File

You can use file input streams (`FileInputStream`, `FileReader`) to read from a file.

```java
import java.io.FileInputStream;
import java.io.IOException;

public class ReadFromFile {
    public static void main(String[] args) {
        try (FileInputStream inputStream = new FileInputStream("myNewFile.txt")) {
            int data;
            while ((data = inputStream.read()) != -1) { // Read byte by byte
                System.out.print((char) data);
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

5.2.5. Closing Streams

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

```java
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.

Important Considerations

- **Closing Streams:** Always close streams using `close()` or try-with-resources to release resources.
- **Character Encoding:** Be mindful of character encoding when dealing with text files (e.g., UTF-8).
- **Other File Operations:** Java provides classes for deleting, renaming, and getting file metadata.
- **Buffered Streams:** For performance optimization, use `BufferedInputStream` and `BufferedOutputStream` to wrap file streams.

5.3. 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.

5.3.1. Overview and Hierarchy

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

- **Foundation:** The `java.util` package contains the core classes and interfaces.
- **Interfaces**: `Collection`, `List`, `Set`, `Map`, etc.
- **Classes**: `ArrayList`, `LinkedList`, `HashSet`, `HashMap`, etc.
- **Hierarchy**:
  - `Collection`: Root interface – represents a group of objects.
    - `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`)

```
Collection
    |
+---List
|       |-- ArrayList
|       |-- LinkedList
|
+---Set
|       |-- HashSet
|
+---Map
        |-- HashMap
```

5.3.2. `ArrayList`

The `ArrayList` class is a resizable [array], which can be found in the `java.util` package.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want.

- 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.

5.3.2.1. Creating an `ArrayList`

```
import java.util.ArrayList; // import the ArrayList class
ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

5.3.2.2. Add Items

The `ArrayList` class has many useful methods. For example, to add elements to the `ArrayList`, use the `add()` method:

```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

5.3.2.3. Access an Item

To access an element in the `ArrayList`, use the `get()` method and refer to the index number:

```
cars.get(0);
```

Remember: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

5.3.2.4. Change an Item

To modify an element, use the `set()` method and refer to the index number:

```
cars.set(0, "Opel");
```

5.3.2.5. Remove an Item

To remove an element, use the `remove()` method and refer to the index number:

```
cars.remove(0);
```

To remove all the elements in the `ArrayList`, use the `clear()` method:

```
cars.clear();
```

5.3.2.6. `ArrayList` Size

To find out how many elements an ArrayList have, use the `size` method:

```
cars.size();
```

5.3.2.7. Loop Through an `ArrayList`

Loop through the elements of an `ArrayList` with a `for` loop, and use the `size()` method to specify how many times the loop should run:

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

You can also loop through an `ArrayList` with the **for-each** loop:

```
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

5.3.2.8. Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: `Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Create an `ArrayList` to store numbers (add elements of type `Integer`):

```java
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

5.3.2.9. Sort an ArrayList

Another useful class in the `java.util` package is the `Collections` class, which include the `sort()` method for sorting lists alphabetically or numerically:

Sort an ArrayList of Strings:

```java
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    Collections.sort(cars);  // Sort cars
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

Sort an ArrayList of Integers:

```java
import java.util.ArrayList;
import java.util.Collections;  // Import the Collections class

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(33);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(34);
    myNumbers.add(8);
    myNumbers.add(12);

    Collections.sort(myNumbers);  // Sort myNumbers

    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
}
```

5.3.3. `LinkedList`

In the previous topic, you learned about the `ArrayList` class. The `LinkedList` class is almost identical to the `ArrayList`.

- 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
// Import the LinkedList class
import java.util.LinkedList;

public class Main {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

5.3.3.1. ArrayList vs. LinkedList

The `LinkedList` class is a collection which can contain many objects of the same type, just like the `ArrayList`.

The `LinkedList` class has all of the same methods as the `ArrayList` class because they both implement the `List` interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the `ArrayList` class and the `LinkedList` class can be used in the same way, they are built very differently.

5.3.3.2. How the ArrayList works

The `ArrayList` class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

5.3.3.3. How the LinkedList works

The `LinkedList` stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

5.3.3.4. When To Use

Use an `ArrayList` for storing and accessing data, and `LinkedList` to manipulate data.

5.3.3.5. LinkedList Methods

For many cases, the `ArrayList` is more efficient as it is common to need access to random items in the list, but the `LinkedList` provides several methods to do certain operations more efficiently:

Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

5.3.4. `HashMap`

In the `ArrayList` topic, you learned that Arrays store items as an ordered collection, and you have to access them with an index number (`int` type). A `HashMap` however, store items in "**key/value**" pairs, and you can access them by an index of another type (e.g. a `String`).

One object is used as a key (index) to another object (value). It can store different types: `String` keys and `Integer` values, or the same type, like: `String` keys and `String` values.

- 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.

Create a `HashMap` object called **capitalCities** that will store `String` **keys** and `String` **values**:

```
import java.util.HashMap; // import the HashMap class

HashMap<String, String> capitalCities = new HashMap<String, String>();
```

5.3.4.1. Add Items

The `HashMap` class has many useful methods. For example, to add items to it, use the `put()` method:

```
// Import the HashMap class
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called capitalCities
    HashMap<String, String> capitalCities = new HashMap<String, String>();

    // Add keys and values (Country, City)
    capitalCities.put("England", "London");
    capitalCities.put("Germany", "Berlin");
    capitalCities.put("Norway", "Oslo");
    capitalCities.put("USA", "Washington DC");
    System.out.println(capitalCities);
  }
}
```

5.3.4.2. Access an Item

To access a value in the `HashMap`, use the `get()` method and refer to its key:

```
capitalCities.get("England");
```

5.3.4.3. Remove an Item

To remove an item, use the `remove()` method and refer to the key:

```
capitalCities.remove("England");
```

To remove all items, use the `clear()` method:

```
capitalCities.clear();
```

5.3.4.4. HashMap Size

To find out how many items there are, use the `size()` method:

```
capitalCities.size();
```

5.3.4.5. Loop Through a HashMap

Loop through the items of a `HashMap` with a **for-each** loop.

Note: Use the `keySet()` method if you only want the keys, and use the `values()` method if you only want the values:

```java
// Print keys
for (String i : capitalCities.keySet()) {
  System.out.println(i);
}
```

```java
// Print values
for (String i : capitalCities.values()) {
  System.out.println(i);
}
```

```java
// Print keys and values
for (String i : capitalCities.keySet()) {
  System.out.println("key: " + i + " value: " + capitalCities.get(i));
}
```

5.3.4.6. Other Types

Keys and values in a HashMap are actually objects. In the examples above, we used objects of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: `Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Create a `HashMap` object called **people** that will store `String` **keys** and `Integer` **values**:

```java
// Import the HashMap class
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    // Create a HashMap object called people
    HashMap<String, Integer> people = new HashMap<String, Integer>();
    // Add keys and values (Name, Age)
    people.put("John", 32);
    people.put("Steve", 30);
    people.put("Angie", 33);
    for (String i : people.keySet()) {
      System.out.println("key: " + i + " value: " + people.get(i));
    }
  }
}
```

5.3.5. `HashSet`

A HashSet is a collection of items where every item is unique, and it is found in the `java.util` package.

- 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.

Create a `HashSet` object called **cars** that will store strings:

```java
import java.util.HashSet; // Import the HashSet class

HashSet<String> cars = new HashSet<String>();
```

5.3.5.1. Add Items

The `HashSet` class has many useful methods. For example, to add items to it, use the `add()` method:

```java
// Import the HashSet class
import java.util.HashSet;

public class Main {
  public static void main(String[] args) {
    HashSet<String> cars = new HashSet<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Note: In the example above, even though BMW is added twice it only appears once in the set because every item in a set has to be unique.

5.3.5.2. Check If an Item Exists

To check whether an item exists in a HashSet, use the `contains()` method:

```java
cars.contains("Mazda");
```

5.3.5.3. Remove an Item

To remove an item, use the `remove()` method:

```java
cars.remove("Volvo");
```

To remove all items, use the `clear()` method:

```java
cars.clear();
```

5.3.5.4. HashSet Size

To find out how many items there are, use the `size` method:

```java
cars.size();
```

5.3.5.5. Loop Through a HashSet

Loop through the items of an `HashSet` with a **for-each** loop:

```
for (String i : cars) {
  System.out.println(i);
}
```

5.3.5.6. Other Types

Items in an HashSet are actually objects. In the examples above, we created items (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: `Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Use a `HashSet` that stores `Integer` objects:

```
import java.util.HashSet;

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

    // Create a HashSet object called numbers
    HashSet<Integer> numbers = new HashSet<Integer>();

    // Add values to the set
    numbers.add(4);
    numbers.add(7);
    numbers.add(8);

    // Show which numbers between 1 and 10 are in the set
    for(int i = 1; i <= 10; i++) {
      if(numbers.contains(i)) {
        System.out.println(i + " was found in the set.");
      } else {
        System.out.println(i + " was not found in the set.");
      }
    }
  }
}
```

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.

5.3.6. Iterator

An `Iterator` is an object that can be used to loop through collections, like ArrayList and HashSet. It is called an "iterator" because "iterating" is the technical term for looping.

To use an Iterator, you must import it from the `java.util` package.

5.3.6.1. Getting an Iterator

The `iterator()` method can be used to get an `Iterator` for any collection:

```java
// Import the ArrayList class and the Iterator class
import java.util.ArrayList;
import java.util.Iterator;

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

    // Make a collection
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    // Get the iterator
    Iterator<String> it = cars.iterator();

    // Print the first item
    System.out.println(it.next());
  }
}
```

5.3.6.2. Looping Through a Collection

To loop through a collection, use the `hasNext()` and `next()` methods of the `Iterator`:

```java
while(it.hasNext()) {
  System.out.println(it.next());
}
```

5.3.6.3. Removing Items from a Collection

Iterators are designed to easily change the collections that they loop through. The `remove()` method can remove items from a collection while looping.

Use an iterator to remove numbers less than 10 from a collection:

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

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(12);
    numbers.add(8);
    numbers.add(2);
    numbers.add(23);
    Iterator<Integer> it = numbers.iterator();
    while(it.hasNext()) {
      Integer i = it.next();
      if(i < 10) {
        it.remove();
      }
    }
    System.out.println(numbers);
  }
}
```

Note: Trying to remove items using a **for loop** or a **for-each loop** would not work correctly because the collection is changing size at the same time that the code is trying to loop.