1. Variables

Description: Variables are used to store data that can be referenced and manipulated within a program.

Example:

```
public class StoreItem {
    public static void main(String[] args) {
        // Declaring variables
        String itemName = "Laptop";
        double price = 899.99;
        int quantity = 5;

        // Display item details
        System.out.println("Item: " + itemName);
        System.out.println("Price: $" + price);
        System.out.println("Quantity in stock: " + quantity);

        // Calculate total value of the items in stock
        double totalValue = price * quantity;
        System.out.println("Total Value: $" + totalValue);
    }
}
```

Explanation: The variables itemName, price, and quantity store the item's name, price, and quantity, respectively. The total value is calculated and printed using the stored data.

2. Arithmetic Operators

Description: Arithmetic operators are used to perform basic arithmetic operations.

Example:

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

```
        int b = 5;

        // Addition
        int sum = a + b;
        System.out.println("Sum: " + sum);

        // Subtraction
        int diff = a - b;
        System.out.println("Difference: " + diff);

        // Multiplication
        int product = a * b;
        System.out.println("Product: " + product);

        // Division
        int quotient = a / b;
        System.out.println("Quotient: " + quotient);

        // Modulo (Remainder)
        int remainder = a % b;
        System.out.println("Remainder: " + remainder);
    }
}
```

Explanation: This example demonstrates basic arithmetic operations like addition, subtraction, multiplication, division, and modulo (remainder) using integers.

3. Conditional Operators (Ternary Operator)

Description: Conditional (ternary) operators evaluate a boolean expression and return one of two values based on the result.

Example:

```
public class DiscountChecker {
```

```
    public static void main(String[] args) {
        double price = 150.00;
        double discount = (price > 100) ? 0.1 : 0.05; // 10% discount if price > 100, else
5%

        double finalPrice = price - (price * discount);
        System.out.println("Original Price: $" + price);
        System.out.println("Discount: " + (discount * 100) + "%");
        System.out.println("Final Price after Discount: $" + finalPrice);
    }
}
```

Explanation: The ternary operator checks if the price is greater than 100. If true, a 10% discount is applied; otherwise, a 5% discount is applied.

4. Logical Operators

Description: Logical operators are used to combine multiple conditions.

Example:

```
public class VotingEligibility {
    public static void main(String[] args) {
        int age = 20;
        boolean hasID = true;

        // Check if a person is eligible to vote
        if (age >= 18 && hasID) {
            System.out.println("You are eligible to vote.");
        } else {
            System.out.println("You are not eligible to vote.");
        }
    }
}
```

Explanation: The logical AND (&&) operator is used to check if both conditions—age being 18 or greater and having an ID—are true for voting eligibility.

5. If-Else

Description: The if-else statement is used to execute different blocks of code based on a condition.

Example:

```
public class AgeGroup {
    public static void main(String[] args) {
        int age = 25;

        if (age < 13) {
            System.out.println("Child");
        } else if (age >= 13 && age < 20) {
            System.out.println("Teenager");
        } else {
            System.out.println("Adult");
        }
    }
}
```

Explanation: The if-else statement checks the person's age and prints whether they are a child, teenager, or adult based on the condition.

6. While Loop

Description: The while loop executes a block of code as long as a specified condition is true.

Example:

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

        // Countdown from 10 to 1
```

```
        while (counter > 0) {
            System.out.println(counter);
            counter--; // Decrement the counter
        }
        System.out.println("Blast off!");
    }
}
```

Explanation: The while loop continues to print numbers from 10 to 1, decrementing the counter on each iteration.

7. Do-While Loop

Description: The do-while loop executes a block of code at least once and then repeats the execution as long as the condition is true.

Example:

```
public class DoWhileExample {
    public static void main(String[] args) {
        int count = 1;

        // Do-while loop to print "Hello" 5 times
        do {
            System.out.println("Hello");
            count++;
        } while (count <= 5);
    }
}
```

Explanation: The do-while loop ensures that the statement "Hello" is printed at least once, and then it repeats until the counter reaches 5.

8. For Loop

Description: The for loop is used for iterating a set number of times.

Example:

```
public class MultiplicationTable {
    public static void main(String[] args) {
        int number = 7;

        // Print multiplication table of 7
        for (int i = 1; i <= 10; i++) {
            System.out.println(number + " x " + i + " = " + (number * i));
        }
    }
}
```

Explanation: The for loop iterates 10 times, printing the multiplication table of the number 7.

9. Break Keyword

Description: The break keyword is used to exit from a loop or switch-case structure prematurely.

Example:

```
public class BreakExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break; // Exit the loop when i reaches 5
            }
            System.out.println(i);
        }
        System.out.println("Loop terminated.");
    }
}
```

Explanation: The loop will print numbers from 1 to 4. When i equals 5, the break statement will terminate the loop.

10. Continue Keyword

Description: The continue keyword is used to skip the current iteration of a loop and move to the next iteration.

Example:

```
public class ContinueExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            if (i % 2 == 0) {
                continue; // Skip even numbers
            }
            System.out.println(i); // Only odd numbers will be printed
        }
    }
}
```

Explanation: The continue statement skips the current iteration for even numbers, printing only odd numbers between 1 and 10.

11. Switch-Case Construct

Description: The switch statement allows a variable to be tested for equality against a list of values, with each value having a corresponding case.

Example:

```
public class WeekdayExample {
    public static void main(String[] args) {
        int day = 3;
        String dayName;

        switch (day) {
            case 1:
                dayName = "Monday";
                break;
            case 2:
```

```
            dayName = "Tuesday";
            break;
        case 3:
            dayName = "Wednesday";
            break;
        case 4:
            dayName = "Thursday";
            break;
        case 5:
            dayName = "Friday";
            break;
        case 6:
            dayName = "Saturday";
            break;
        case 7:
            dayName = "Sunday";
            break;
        default:
            dayName = "Invalid day";
            break;
    }

    System.out.println("The day is: " + dayName);
  }
}
```

Explanation: The switch statement checks the value of day and prints the corresponding weekday. The break statement ensures that only one case block is executed.

These examples demonstrate how each of the Java concepts is applied in real-world scenarios, from arithmetic operations to controlling the flow of logic using loops and conditional statements.

Loops in Java

A **loop** is a fundamental programming concept that allows you to execute a block of code repeatedly based on a given condition. In Java, there are three primary types of loops:

1. **For Loop**
2. **While Loop**
3. **Do-While Loop**

Additionally, Java provides **enhanced for loop** (also known as **for-each loop**) to iterate over arrays and collections.

1. For Loop

The for loop is the most commonly used loop in Java. It is typically used when the number of iterations is known beforehand.

Syntax:

```
for (initialization; condition; update) {
    // code to be executed
}
```

- **Initialization**: Executed once at the beginning of the loop.
- **Condition**: The condition is evaluated before each iteration. If it evaluates to true, the loop continues; otherwise, it terminates.
- **Update**: Executed after each iteration to update the loop variable.

Example:

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

This will output:

0

1

2

3

4

2. While Loop

The while loop is used when the number of iterations is not necessarily known ahead of time, and you want the loop to continue as long as a condition is true.

Syntax:

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

- The condition is checked before each iteration. If it is true, the loop executes; otherwise, it terminates.

Example:

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

This will output:

```
0
1
2
3
4
```

3. Do-While Loop

The do-while loop is similar to the while loop, but with one key difference: it guarantees that the block of code will be executed **at least once**, even if the condition is false from the start.

Syntax:

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

- The code inside the loop is executed first, and then the condition is checked. If the condition is true, the loop runs again.

Example:

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

This will output:

```
0
1
2
3
4
```

4. Enhanced For Loop (For-each Loop)

The **enhanced for loop** is a special loop in Java designed for iterating over arrays or collections (like lists and sets). It simplifies the process of accessing each element of a collection or array.

Syntax:

```
for (datatype variable : arrayOrCollection) {
    // code to be executed
}
```

- **datatype**: The type of the variable (e.g., int, String).
- **variable**: The element being iterated over in each cycle of the loop.
- **arrayOrCollection**: The array or collection you're iterating over.

Example (Array):

```
int[] arr = {1, 2, 3, 4, 5};
for (int num : arr) {
    System.out.println(num);
}
```

This will output:

```
1
```

2

3

4

5

Example (List):

```
List<String> list = Arrays.asList("apple", "banana", "cherry");
for (String fruit : list) {
    System.out.println(fruit);
}
```

This will output:

apple

banana

cherry

Breaking and Continuing in Loops

- **Break Statement**: The break statement can be used to exit a loop prematurely, even if the loop's condition has not been met.

Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
```

This will output:

0

1

2

3

4

- **Continue Statement**: The continue statement can be used to skip the current iteration and proceed to the next one, bypassing the remaining code for the current loop iteration.

Example:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue;  // Skip printing 5
    }
    System.out.println(i);
}
```

This will output:

0

1

2

3

4

6

7

8

9

Nested Loops

You can place loops inside other loops, which is called **nested loops**. The inner loop executes completely for every iteration of the outer loop.

Example (Nested for loop):

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 2; j++) {
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

This will output:

i = 0, j = 0

i = 0, j = 1

i = 1, j = 0

i = 1, j = 1

i = 2, j = 0

i = 2, j = 1

Best Practices for Using Loops

1. **Avoid Infinite Loops**: Ensure that the loop's condition eventually becomes false. For example, always update the loop variable in a while loop.
2. **Choose the Right Loop Type**:
    - Use for loops when the number of iterations is known.
    - Use while loops when the number of iterations is unknown and dependent on a condition.
    - Use do-while loops when you want the code to run at least once.
    - Use enhanced for loops to simplify iterating through arrays and collections.
3. **Limit Nested Loops**: Deeply nested loops can be inefficient. Consider optimizing the algorithm if you need many nested loops.

Sure! Below are detailed notes on **String**, **StringBuffer**, and **StringBuilder** in Java:

1. String in Java

A **String** is an object that represents a sequence of characters in Java. It is widely used in Java programming for text manipulation.

Key Characteristics:

- **Immutable**: Once a String object is created, its value cannot be changed. Any modification to a String results in a new String object being created.
- **String Pool**: Java maintains a special memory area known as the **String Pool** or **String Constant Pool** to optimize memory usage for strings. When you create a string literal (e.g., "hello"), it checks if the string already exists in the pool. If it does, it reuses the existing object; otherwise, a new object is created.

Syntax:

String str = "Hello, World!";

Operations:

- **Concatenation**: You can concatenate strings using the + operator or the concat() method.
- String greeting = "Hello, " + "World!"; // Using +
- String message = "Hello, ".concat("World!"); // Using concat()
- **Comparison**: To compare strings, you should use the .equals() method, not ==, because == checks for reference equality, whereas .equals() checks for content equality.
- String str1 = "Java";
- String str2 = "Java";
- boolean isEqual = str1.equals(str2); // true
- **Immutability**: If you modify a String, a new object is created.
- String str = "Hello";
- str = str + " World"; // Creates a new String object

Common Methods:

- length(): Returns the length of the string.
- int len = str.length();
- charAt(index): Returns the character at the specified index.
- char ch = str.charAt(0); // 'H'
- substring(start, end): Returns a substring from start index to end index.
- String sub = str.substring(0, 5); // "Hello"
- toLowerCase(), toUpperCase(): Converts the string to lowercase or uppercase.
- String lower = str.toLowerCase();
- trim(): Removes leading and trailing whitespace.
- String trimmed = " Hello ".trim(); // "Hello"

2. StringBuffer in Java

StringBuffer is a mutable sequence of characters. Unlike String, StringBuffer allows you to modify the content of the string without creating new objects. It is generally used when you need to perform a lot of modifications on strings.

Key Characteristics:

- **Mutable**: You can change the content of a StringBuffer without creating a new object.
- **Thread-Safe**: StringBuffer is synchronized, meaning it is thread-safe (useful for multi-threaded programs).
- **Efficient for frequent modifications**: StringBuffer is more efficient than String when performing repeated string modifications (e.g., concatenation).

Syntax:

StringBuffer sb = new StringBuffer("Hello");

Common Methods:

- **append()**: Adds content to the end of the StringBuffer.
- sb.append(" World"); // "Hello World"
- **insert()**: Inserts content at a specific position.
- sb.insert(5, ","); // "Hello, World"
- **delete()**: Removes content from the StringBuffer between the specified indices.
- sb.delete(5, 6); // "Hello World" -> "HelloWorld"
- **reverse()**: Reverses the content of the StringBuffer.
- sb.reverse(); // "World, Hello"
- **replace()**: Replaces content between the specified indices with a new string.
- sb.replace(5, 11, "Java"); // "Hello Java"

Example:

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

sb.insert(5, ",");

System.out.println(sb); // Output: "Hello, World"

3. StringBuilder in Java

StringBuilder is similar to StringBuffer but **not synchronized**. This means it is **not thread-safe**, but it performs better in scenarios where thread safety is not a concern. It is used when you need to modify the string content in a single-threaded environment.

Key Characteristics:

- **Mutable**: Like StringBuffer, StringBuilder allows modification of the string without creating new objects.
- **Not Thread-Safe**: It is faster than StringBuffer because it does not involve synchronization.
- **Efficient for frequent modifications in single-threaded programs**.

Syntax:

StringBuilder sb = new StringBuilder("Hello");

Common Methods:

- **append()**: Adds content to the end of the StringBuilder.
- sb.append(" World"); // "Hello World"
- **insert()**: Inserts content at a specific position.
- sb.insert(5, ","); // "Hello, World"
- **delete()**: Removes content between the specified indices.
- sb.delete(5, 6); // "Hello World" -> "HelloWorld"
- **reverse()**: Reverses the content of the StringBuilder.
- sb.reverse(); // "World, Hello"
- **replace()**: Replaces content between the specified indices with a new string.
- sb.replace(5, 11, "Java"); // "Hello Java"

Example:

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

sb.insert(5, ",");

System.out.println(sb); // Output: "Hello, World"

Differences between String, StringBuffer, and StringBuilder

Feature	**String**	**StringBuffer**	**StringBuilder**
Immutability	Immutable	Mutable	Mutable
Thread-Safety	Not synchronized	Synchronized (thread-safe)	Not synchronized (not thread-safe)
Performance	Slower for frequent modifications	Slower due to synchronization	Faster (because not synchronized)

Feature	String	StringBuffer	StringBuilder
Use Case	Text manipulation with minimal changes	Frequent modifications in multi-threaded programs	Frequent modifications in single-threaded programs
Common Methods	append(), equals(), substring()	append(), insert(), reverse()	append(), insert(), reverse()

When to Use Each

- **Use String** when you do not need to modify the string frequently, especially when the string values are fixed or constant.
- **Use StringBuffer** if you need a mutable sequence of characters and you are working in a multi-threaded environment where thread safety is a concern.
- **Use StringBuilder** if you need a mutable sequence of characters but you are working in a single-threaded environment, as it is faster than StringBuffer.