3.1 Whitespace, Identifiers, Literals, Comments, Separators and Keywords

In Java, there are several elements that constitute the syntax of the language. These include whitespace, literals, comments, separators, and keywords. Here's an explanation of each:

1. **Whitespace**:
   - Whitespace refers to spaces, tabs, and newlines in a Java program.
   - It is used to separate tokens (such as keywords, identifiers, operators, etc.) to make the code more readable.
   - Whitespace is ignored by the Java compiler.
   - Example

     ```
     int num1 = 5;
     int num2=10; // Whitespace is not required, but it makes the code more readable
     ```

2. **Literals**:
   - Literals are data values that are represented directly in the code.
   - There are different types of literals in Java:
     - **Integer literals**: e.g., 10, 255, 0xFF (hexadecimal), 077 (octal).
     - **Floating-point literals**: e.g., 3.14, 2.0e3.
     - **Character literals**: e.g., 'A', '\n'.
     - **String literals**: e.g., "Hello, World!".
     - **Boolean literals**: true or false.
     - **Null literal**: null.
   - Example:

     ```
     int integerValue = 42; // Integer literal
     double doubleValue = 3.14; // Floating-point literal
     char charValue = 'A'; // Character literal
     String stringValue = "Hello, World!"; // String literal
     boolean boolValue = true; // Boolean literal
     ```

3. **Comments**:
   - Comments are used to add explanatory notes to the code.
   - They are ignored by the compiler and do not affect the program's behavior.
   - There are two types of comments in Java:
     - **Single-line comments**: These begin with // and continue until the end of the line.
     - **Multi-line comments**: These are enclosed between /* and */ and can span multiple lines.
   - Example

     ```
     // This is a single-line comment
     /*
      * This is a multi-line comment.
      * It can span multiple lines.
      */
     ```
4. **Separators**:
   - Separators are characters used to separate tokens in the code.
   - Common separators in Java include:
     - Semicolon ;: Used to terminate statements.

- Comma ,: Used to separate elements in a list.
- Parentheses (): Used for grouping and method parameters.
- Curly braces {}: Define code blocks (used in methods, classes, loops, etc.).
- Square brackets []: Used for array declarations and indexing.
- Period .: Used for member access (e.g., object.method()).
- Arrow ->: Used in lambda expressions.

- Example:

```
int[] numbers = {1, 2, 3}; // Curly braces {} to define an array
String fullName = "John Doe"; // Quotation marks "" for strings
myObject.myMethod(); // Period . to access a method or member
```

5. **Keywords**:
   - Keywords are reserved words in Java that have a special meaning and cannot be used as identifiers (like variable or method names).
   - Some examples of Java keywords include public, class, int, if, else, while, for, return, static, void, new, extends, implements, etc.
   - There are 50 keywords defined in the java language.

abstarct	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

   - Example:

```
public class Example { // 'public' is a keyword for class visibility
    public static void main(String[] args) { // 'public', 'static', 'void' are keywords
        int number = 10; // 'int' is a keyword for integer type
        if (number > 5) { // 'if' is a keyword for conditional statements
            System.out.println("Number is greater than 5"); // 'System', 'out', 'println' are
keywords
        }
    }
}
```

For example, consider the following Java code snippet:

```
// This is a single-line comment

public class HelloWorld {
    public static void main(String[] args) {
        int number = 10; // This is an integer literal
        String message = "Hello, World!"; // This is a string literal

        if (number > 5) {
            System.out.println(message);
        }
    }
}
```

In this example, you can see the use of whitespace, literals (integer and string), single-line comments, separators (such as semicolons and curly braces), and keywords (such as public, class, int, if, else, etc.).

3.2 Data Types and Conversion

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = **false**

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, **byte** b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: short s = 10000, **short** r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: int a = 100000, **int** b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, **long** b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example: float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example: char letterA = 'A'

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system (universal international standard character encoding that is capable of representing most of the world's written languages.) not ASCII code system. The \u0000 is the lowest range of Unicode system.

3.3 Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java:

1) Local Variable

- Local variables are declared within a method, constructor, or block.
- They are only accessible within the scope in which they are defined.
- They do not have a default value and must be initialized before use.
- Example:

```
public void myMethod() {
   int localVar = 10; // localVar is a local variable
   System.out.println(localVar);
}
```

2) Instance Variable

- Instance variables are declared within a class, but outside of any method, constructor, or block.
- They belong to the object and each instance of the class has its own copy.
- They are initialized with default values (0, false, or null) if not explicitly set.
- Example:

```
public class MyClass {
    int instanceVar; // instanceVar is an instance variable
    public void setInstanceVar(int value) {
        this.instanceVar = value;
    }
}
```

3) Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

```
public class MyClass {
    static int classVar = 100; // classVar is a class variable
    public static void main(String[] args) {
        System.out.println(classVar);
    }
}
```

3.4 Constants

Constant refers to a variable whose value cannot be changed after it has been initialized. This means that once a constant is assigned a value, it cannot be modified during the program's execution.

To create a constant in Java, you use the `final` keyword. Here's how you define a constant:

```
final dataType CONSTANT_NAME = value;
```

Example:

```
final int MAX_VALUE = 100;
```

In this example, `MAX_VALUE` is a constant of type `int` with a value of `100`. Since it's declared with the `final` keyword, you cannot change its value later in the program.

Constants are often used to define values that are not expected to change during the execution of a program, such as mathematical constants (e.g., pi), configuration settings, or any value that is supposed to remain fixed.

It's also a common practice to name constants in all uppercase letters to distinguish them from regular variables. For example, `MAX_VALUE` instead of `maxValue`.

3.5 Operators

Operators are symbols that perform operations on variables and values.
Operators in Java can be classified into 5 types:

1. Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

2. Assignment Operators

Assignment operators are used in Java to assign values to variables.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

3. Relational/Comparison Operators

Relational operators are used to check the relationship between two operands.

Operator	Description	Example
==	Is Equal To	3 == 5 returns **false**
!=	Not Equal To	3 != 5 returns **true**
>	Greater Than	3 > 5 returns **false**
<	Less Than	3 < 5 returns **true**
>=	Greater Than or Equal To	3 >= 5 returns **false**
<=	Less Than or Equal To	3 <= 5 returns **true**

4. Logical Operators

Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 **&&** expression2	true only if both expression1 and expression2 are true
|| (Logical OR)	expression1 || expression2	true if either expression1 or expression2 is true
! (Logical NOT)	**!**expression	true if expression is false and vice versa

5. Unary Operators

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by **1**. That is, ++5 will return **6**.

Different types of unary operators are:

Operator	Meaning
+	**Unary plus**: not necessary to use since numbers are positive without using it
-	**Unary minus**: inverts the sign of an expression
++	**Increment operator**: increments value by 1
--	**Decrement operator**: decrements value by 1
!	**Logical complement operator**: inverts the value of a boolean

6. Bitwise Operators

Bitwise operators in Java are used to perform operations on individual bits.

Operator	Description
~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR

Java Ternary Operator

The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example,

variable = Expression ? expression1 : expression2

Here's how it works.

- If the Expression is true, expression1 is assigned to the variable.
- If the Expression is false, expression2 is assigned to the variable.

3.6 Strings

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. The java.lang.String class is used to create a string object. For example:

```
char[] ch={'p','o','k','h','a','r','a'};
String s=new String(ch);
```

is same as: String s="pokhara";

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

There are two ways to create String object:

1. *By string literal*

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

2. *By new keyword*

```
String s=new String("Welcome");//creates two objects and one reference variable
```

Example:

```
public class StringExample {
   public static void main(String args[]) {
      String s1 = "java"; // creating string by Java string literal
      char ch[] = {'s', 't', 'r', 'i', 'n', 'g', 's'};
      String s2 = new String(ch); // converting char array to string
      String s3 = new String("example"); // creating Java string by new keyword

      System.out.println(s1);
      System.out.println(s2);
      System.out.println(s3);
   }
}
```

Some String Methods

The `String` class provides a wide range of methods for manipulating strings, such as:

- `length()`: Returns the length of the string.
- `charAt(int index)`: Returns the character at the specified index.
- `substring(int beginIndex)`: Returns a substring from the specified index.
- `indexOf(String str)`: Returns the index of the first occurrence of a substring.
- `toUpperCase()` and `toLowerCase()`: Converts the string to uppercase or lowercase, respectively.

3.7 Control Structures

In Java, control structures are used to control the flow of execution in a program. They allow you to make decisions, loop over a set of instructions, and execute code based on conditions. Here are the main control structures in Java:

Java provides three types of control flow statements.

1. Decision Making statements
   - if statements
   - switch statement
2. Loop statements
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. Jump statements
   - break statement
   - continue statement

Decision-Making statements:

Decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
statement 1; //executes when condition is true
}
```

Example:

Student.java

```
public class Student {
   public static void main(String[] args) {
      int x = 10;
      int y = 12;
      if(x + y > 20) {
         System.out.println("x + y is greater than 20");
      }
   }
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
        statement 1; //executes when condition is true
}
else{
        statement 2; //executes when condition is false
}
```

Consider the following example.
Student.java

```
public class Student {
   public static void main(String[] args) {
      int x = 10;
      int y = 12;
      if(x + y < 10) {
         System.out.println("x + y is less than 10");
      } else {
         System.out.println("x + y is greater than 20");
      }
   }
}
```

Output:

```
x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
        statement 1; //executes when condition 1 is true
}
else if(condition 2) {
        statement 2; //executes when condition 2 is true
}
else {
        statement 2; //executes when all the conditions are false
}
```

Consider the following example.

Student.java

```
public class Student {
    public static void main(String[] args) {
        String city = "Delhi";
        if(city == "Meerut") {
            System.out.println("city is meerut");
        } else if (city == "Noida") {
            System.out.println("city is noida");
        } else if(city == "Agra") {
            System.out.println("city is agra");
        } else {
            System.out.println(city);
        }
    }
}
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
            statement 1; //executes when condition 1 is true
            if(condition 2) {
                        statement 2; //executes when condition 2 is true
            }
            else{
                        statement 2; //executes when condition 2 is false
            }
}
```

Consider the following example.

Student.java

```
public class Student {
    public static void main(String[] args) {
        String address = "Delhi, India";

        if(address.endsWith("India")) {
            if(address.contains("Meerut")) {
               System.out.println("Your city is Meerut");
            } else if(address.contains("Noida")) {
               System.out.println("Your city is Noida");
            } else {
               System.out.println(address.split(",")[0]);
            }
        } else {
            System.out.println("You are not living in India");
        }
    }
}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
    case value1:
              statement1;
     break;
    .
     case valueN:
             statementN;
     break;
    default:
             default statement;
}
```

Example:

Student.java

```
public class Student implements Cloneable {
    public static void main(String[] args) {
        int num = 2;
        switch (num){
            case 0:
                System.out.println("number is 0");
                break;
            case 1:
                System.out.println("number is 1");
                break;
            default:
                System.out.println(num);
        }
    }
}
```

Output:

```
2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

3.8 Loop

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

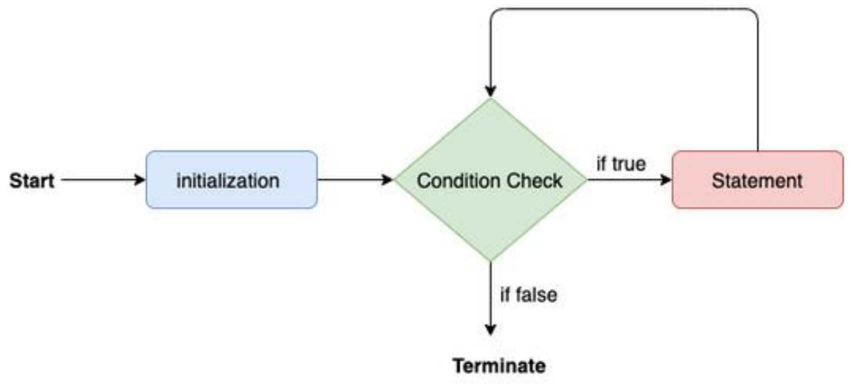
1. for loop
2. while loop
3. do-while loop

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
            //block of statements
}
```

The flow chart for the for-loop is given below.



Example:

Calculation.java

```
public class Calculattion {
    public static void main(String[] args) {
        int sum = 0;
        for(int j = 1; j<=10; j++) {
            sum = sum + j;
        }
        System.out.println("The sum of first 10 natural numbers is " + sum);
    }
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name){
            //statements
}
```

Calculation.java

```
public class Calculation {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String[] names = {"Java","C","C++","Python","JavaScript"};
        System.out.println("Printing the content of the array names:\n");
        for(String name:names) {
            System.out.println(name);
        }
    }
}
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

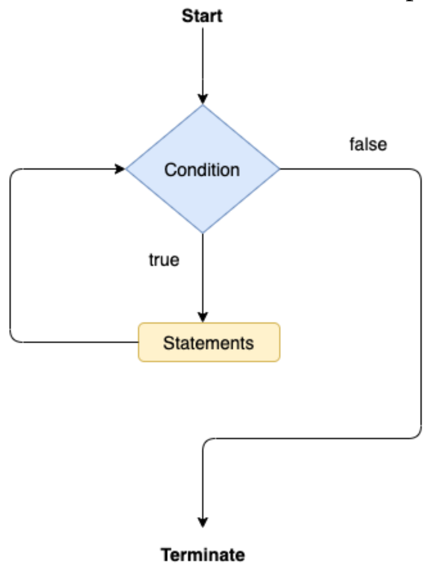
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
while(condition){
        //looping statements
}
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
public class Calculation {
   public static void main(String[] args) {
      // TODO Auto-generated method stub
      int i = 0;
      System.out.println("Printing the list of first 10 even numbers \n");
      do {
         System.out.println(i);
         i = i + 2;
      } while(i<=10);
   }
}
```

Output:

```
Printing the list of first 10 even
numbers
0
2
4
6
8
10
```

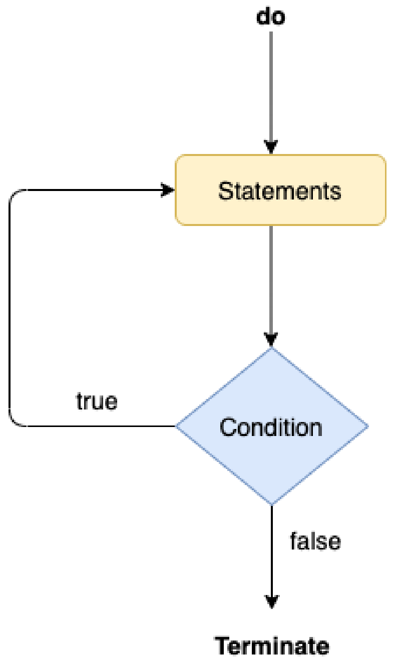
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

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

The flow chart of the do-while loop is given in the following image.



Calculation.java

```
public class Calculation {
   public static void main(String[] args) {
      // TODO Auto-generated method stub
      int i = 0;
      System.out.println("Printing the list of first 10 even numbers \n");
      do {
         System.out.println(i);
         i = i + 2;
      } while(i<=10);
   }
```

Output:

```
Printing the list of first 10 even
numbers
0
2
4
6
8
10
```

```
}
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```
public class BreakExample {
    public static void main(String[] args) {
        for(int i = 0; i<= 10; i++) {
            System.out.println(i);
            if(i==6) {
                break;
            }
        }
    }
}
```

Output:

```
0
1
2
3
4
5
6
```

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {
    public static void main(String[] args) {
        for(int i = 0; i <= 2; i++) {
            for (int j = i; j <= 5; j++) {
                if(j == 4) {
                    continue;
                }
                System.out.println(j);
            }
        }
    }
}
```

Output:

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

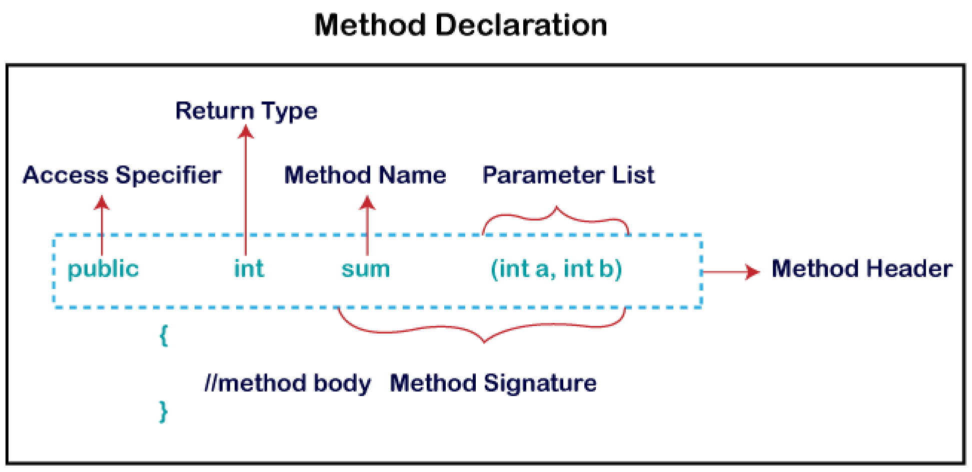
These control structures provide the means to make decisions, repeat blocks of code, and alter the flow of a program based on conditions. They are fundamental for creating complex, dynamic, and interactive programs.

3.9 Methods

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Example:

Demo.java

```
public class Demo
{
        public static void main(String[] args)
        {
                // using the max() method of Math class
                System.out.print("The maximum number is: " + Math.max(9,7));
        }
}
```

Output:

```
The maximum number is: 9
```

User-defined Method

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

Creating a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method
public static void findEvenOdd(int num)
{
        //method body
        if(num%2==0)
                System.out.println(num+" is even");
        else
                System.out.println(num+" is odd");
}
```

We have defined the above method named findevenodd(). It has a parameter **num** of type int. The method does not return any value that's why we have used void. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number **is even**, else prints the number **is odd**.

Calling a Method

In the above example, we have declared a method named findEvenOdd (). Now, to use the method, we need to call it.

Here's is how we can call the findEvenOdd () method.

```
// calls the method
findEvenOdd();
```

Method Return Type

A Java method may or may not return a value to the function call. We use the **return statement** to return any value. For example,

```
int addNumbers() {
...
return sum;
}
```

Here, we are returning the variable *sum*. Since the return type of the function is int. The sum variable should be of int type. Otherwise, it will generate an error.

Example:

```
class Main {

// create a method
  public static int square(int num) {

    // return statement
    return num * num;
  }

  public static void main(String[] args) {
    int result;

    // call the method
    // store returned value to result
    result = square(10);

    System.out.println("Squared value of 10 is: " + result);
  }
}
```

Output:

```
Squared value of 10 is: 100
```

In the above program, we have created a method named square(). The method takes a number as its parameter and returns the square of the number.

Here, we have mentioned the return type of the method as int. Hence, the method should always return an integer value.

Note: If the method does not return any value, we use the void keyword as the return type of the method. For example,

```
public void square(int a) {
  int square = a * a;
  System.out.println("Square is: " + square);
}
```

3.10 Arrays

Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Single Dimensional Array in Java:

Syntax to Declare an Array in Java

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example:

```
//Java Program to illustrate how to declare, instantiate, initialize  and traverse the Java array.
class Testarray{
        public static void main(String args[]){
                int a[]=new int[5];//declaration and instantiation
                a[0]=10;//initialization
                a[1]=20;  a[2]=70;
                a[3]=40;  a[4]=50;
                //traversing array
                for(int i=0;i<a.length;i++)//length is the property of array
                        System.out.println(a[i]);
}  }
```

```
Output:
10
20
70
40
50
```

Example:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
        System.out.println(a[i]);
```

For-each Loop for Java Array
We can also print the Java array using **for-each loop**.
Syntax:

```
        for(data_type variable:array){

        //body of the loop

        }
```

Example:

```
class Testarray1{
        public static void main(String args[]){
                int arr[]={33,34,44,56};
                //printing array using for-each loop
                for(int i:arr)
                System.out.println(i);
        }
}
```

Output:

```
        33
        34
        43
        56
```

Multidimensional Array in Java:

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
        dataType[][] arrayRefVar; (or)
        dataType [][]arrayRefVar; (or)
        dataType arrayRefVar[][]; (or)
        dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
	public static void main(String args[]){

		//declaring and initializing 2D array
		int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

		//printing 2D array
		for(int i=0;i<3;i++){
			for(int j=0;j<3;j++){
				System.out.print(arr[i][j]+" ");
			 }
		System.out.println();
		}
	}
}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.
public class TestArrayException{
	public static void main(String args[]){
		int arr[]={50,60,70,80};
		for(int i=0;i<=arr.length;i++){
```

```
                    System.out.println(arr[i]);
            }
}}
```

Array Copying
We can assign one array variable to another, but then both variables refer to same array
Example
int[]arr = {1,2,3,4};
Int[] arr2 = arr1; // same as arr 1

If we actually want to copy all values of one array into a new array, we use the copyOf method in the Arrays class (**java.util.Arrays**)

Example:

    Int[] copyOfArr1 = Arrays.copyOf (arr1, size);

/*size defines the actual size for new array **copyOfArr** 1 this could be of same, less or more than that of size of arr 1 If size is more than that of arr 1 then additional elements will be 0 (for integers), false(for booleans null (for reference types) If size is less than that of arr 1 then only the initial values are copied. */