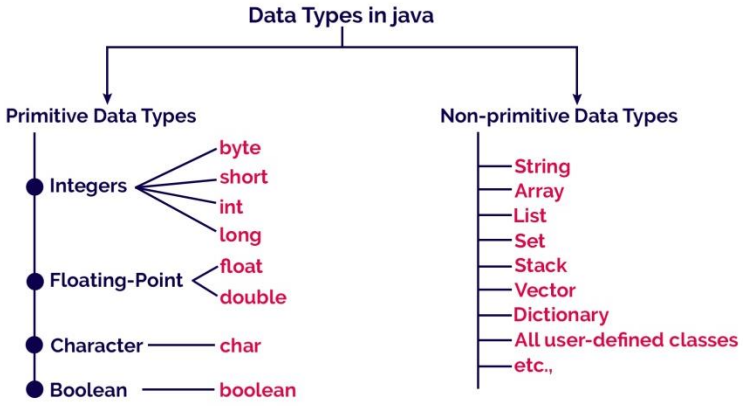
UNIT-2

Primitive Data types

Java programming language has a rich set of data types. The data type is a category of data stored in variables. In java, data types are classified into two types and they are as follows.

- Primitive Data Types
- Non-primitive Data Types



The primitive data types are built-in data types and they specify the type of value stored in a variable and the memory size. The primitive data types do not have any additional methods

The following table provides more description of each primitive data type.

Data type	Meaning	Memory size	Range	Default Value
Integer				
byte	Whole numbers	1 byte	-128 to +127	0
short	Whole numbers	2 bytes	-32768 to +32767	0
int	Whole numbers	4 bytes	-2,147,483,648 to +2,147,483,647	0
long	Whole numbers	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0L
float	Fractional numbers	4 bytes	3.4e-038 to 3.4e+038 Storing 6 to 7 decimal digits	0.0f
Floating Point				
double	Fractional numbers	8 bytes	1.7e-308 to 1.7e+038 sufficient for storing 15 decimal	0.0d
char	Single character	2 bytes	Store single character or letter	\u0000
Boolean				
boolean	unsigned char	1 bit	Stores true or false values	false

User Defined Data Types:

Developers can create their custom data types in Java using **classes** and ***interfaces***. These user-defined data types allow for creating objects with **specific attributes and behaviors**, making Java a **versatile** and **object-oriented** programming language.

```
Student obj = new Student();
```

Here **obj** is a variable of data type **Student** and we call them reference variables as they can be used to store the reference to the object of that class.

Declaration of Variables and Assignment

In Java, variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things:

- It tells the compiler what the variable name is.
- It specifies what type of data the variable will hold.
- The place of declaration (in the program) decides the scope of the variable.

A variable can be used to store a value of any data type. That is, the name has nothing to do with the type. Java allows any properly formed variable to have any declared data type.

Declaration of Variables:

You declare variables by specifying the data type and the variable name. Here are some examples:

- **Primitive Types:**

```
int age;         // Declaration of an integer variable named 'age'
double salary;   // Declaration of a double variable named 'salary'
char grade;      // Declaration of a char variable named 'grade'
boolean isStudent;  // Declaration of a boolean variable named 'isStudent'
```

- **Reference Types:**

```
String name;     // Declaration of a String variable named 'name'
MyClass myObject; // Declaration of a variable of a custom class named
'myObject'
```

Assignment of Values:

After declaring a variable, you can assign a value to it using the assignment operator (=):

```
int age = 25;         // Declaration and assignment for an integer variable
double salary = 50000.75;  // Declaration and assignment for a double variable
```

```
char grade = 'A';           // Declaration and assignment for a char variable
boolean isStudent = true;   // Declaration and assignment for a boolean variable
String name = "John Doe";   // Declaration and assignment for a String variable
MyClass myObject = new MyClass();// Declaration and assignment for a custom class
variable
```

Constant:
A **constant** is an entity in programming that is immutable. In other words, the value that cannot be changed.
Java **does not directly support the constants**. There is an alternative way to define the constants in Java by using the non-access modifiers **static** and **final**.

- The purpose to use the **static modifier** is to manage the memory.
- It also allows the variable to be available **without loading any instance of the class** in which it is defined.
- The **final modifier** represents that the value of the variable **cannot be changed**. It also makes the primitive data type immutable or unchangeable.

The syntax to declare a constant is as follows:

```
static final datatype identifier_name=value;
```

Example:

```
static final double PRICE=432.78;
```

Identifier:
All Java variables must be identified with **unique names**.These unique names are called **identifiers**.
Identifiers can be **short names** (like x and y) or **more descriptive** names (age, sum, totalVolume).

The general rules for naming variables are:

- Names can contain letters, digits, underscores, and dollar signs
- Names must begin with a letter
- Names should start with a lowercase letter and it cannot contain whitespace
- Names can also begin with $ and _
- 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

Example: Valid Identifier

```
/ Variables
int age;
double salaryAmount;
String firstName;
```

```
// Classes
class Car {

}
class StudentDetails {

}
// Methods
void printDetails() {

}
int calculateSum(int num1, int num2) {

}
// Objects
Car myCar = new Car();
StudentDetails student = new StudentDetails();
// Constants
final double PI = 3.14159;
final String GREETING = "Hello";
// Packages
package com.example.myapp;
```

Example of Invalid Identifier:

```
// Variables
int 1stNumber;         // starts with a digit - invalid
double salary-amount;   // contains a hyphen - invalid
String first Name;      // contains a space - invalid

// Classes
class My@Car {          // contains a special character - invalid
class 123Class {        // starts with a digit - invalid

// Methods
void print Details() {  // contains a space - invalid
int calculate Sum(int num1, int num2) {  // space in method name - invalid

// Objects
Car my-Car = new Car();          // contains a hyphen - invalid
StudentDetails 123student = new StudentDetails();  // starts with a digit -
invalid

// Constants
final double Pi = 3.14159;      // case-sensitive, not recommended
```

```
final String greeting_message = "Hello";  // underscores are allowed but not
recommended

// Packages
package com.example.my app;  // contains a space - invalid
```

Literals:

In Java, **literals** are the constant values that appear directly in the program. It can be assigned directly to a variable. Java has various types of literals. The following figure represents a literal.

Here are some common types of literals in Java:

Integer Literals:
Example: int number = 42;
In this case, 42 is an integer literal.

Floating-Point Literals:
Example: double pi = 3.14;
In this case, 3.14 is a floating-point literal.

Boolean Literals:
Example: boolean flag = true;
true and false are boolean literals.

Character Literals:
Example: char grade = 'A';
In this case, 'A' is a character literal.

String Literals:
Example: String message = "Hello, World!";
"Hello, World!" is a string literal.

Null Literal:
Example: Object obj = null;
null is a special literal representing the absence of a value.

Underscores in Numeric Literals (Java 7 and later):

Example: long bigNumber = 1_000_000_000;

The underscores improve readability in large numeric literals.

Type Conversion and Casting:

The process of converting a value from one data type to another is known as **type conversion in Java**. Type conversion is also known as type casting in Java or simply **'casting'**.

Types of Casting in Java

Two types of casting are possible in Java are as follows:

- Implicit type casting (also known as automatic type conversion)
- Explicit type casting

Implicit type casting:

Automatic conversion (casting) done by Java compiler internally is called **implicit conversion** or **implicit type casting** in java.

Implicit casting is performed to convert a **lower data type into a higher data type**. It is also known as **automatic type promotion** in Java.

```
int x = 20;
long y = x; // Automatic conversion
byte z = x;  // Type mismatch: cannot convert from int to byte.
```

Explicit Type Casting:

No automatic conversion will take place from double to byte. These kinds of conversions can be done by using a technique called **explicit casting**.

Type casting performs an explicit conversion between **incompatible types**. Therefore, it is also known as an **explicit type casting** in Java. It is used to convert an **object or variable of one type to another**.

It must be done explicitly by the **programmer**.

```
double d = 100.9;
long l = (long)d; // Explicit type casting.
```

Variables Definition and Assignment in Java

In Java, variable definition and assignment are essential for storing and manipulating data within a program.

Defining Variables:

Syntax:

```
data_type variable_name;
```

example:

```
int myNumber;
double myDouble;
String myString;
```

Assigning Variables:

Syntax:

```
datatype variable_name = value;
```

example:

```
int myNumber = 42;
double myDouble = 3.14;
String myString = "Hello, Java!";
```

Array of Primitive data types:

```
int[] intArray = new int[5];
double[] doubleArray = new double[3];
boolean[] booleanArray = new boolean[4];
char[] charArray = new char[6];
byte[] byteArray = new byte[8];
short[] shortArray = new short[10];
long[] longArray = new long[7];
float[] floatArray = new float[4];
```

Java Comment

The Java comments are the statements in a program that are not executed by the compiler and interpreter.

- Comments are used to make the program more readable by adding the details of the code.
- It makes easy to maintain the code and to find the errors easily.
- The comments can be used to provide information or explanation about the variable, method, class, or any statement.
- It can also be used to prevent the execution of program code while testing the alternative code.

There are three types of comments in Java.

- Single Line Comment
- Multi Line Comment
- Documentation Comment

Single Line Comment:

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements.

Single line comments starts with two forward slashes (//). Any text in front of // is not executed by Java.

Example:

```
//Single line comment
```

Multi Line Comment:
The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there).
Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

Example:

```
/*
This is multi line
comment
*/
```

Documentation Comment:
Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API.
We need to use the **javadoc tool**. The documentation comments are placed between /** and */.

```
/**
*
*We can use various tags to depict the parameter
*or heading or author name
*We can also use HTML tags
*
*/
```

Garbage Collection in java:

Garbage collection in Java is the process by which Java programs perform **automatic memory management**. Java programs **compile to bytecode** that can be run on a **Java Virtual Machine**, or JVM for short. When Java programs run on the JVM, **objects are created on the heap**, which is a portion of **memory dedicated** to the program. Eventually, **some objects will no longer be needed**. The garbage collector finds these unused objects and deletes them to **free up memory**.

In C/C++, a programmer is responsible for both the creation and destruction of objects. Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing **OutOfMemoryErrors**.

But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**

Advantages of Garbage Collection:

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

Operators in Java:

An operator, in java, it is a special symbolic performing specific operations on one, two or three operands and the returning a result.

The Arithmetic Operator
Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators –
Assume integer variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

Example:

```
public class App {
    public static void main(String[] args) throws Exception {
        int a = 10;
        int b = 20;
        System.out.println("a + b = " + (a + b) );
        System.out.println("a - b = " + (a - b) );
        System.out.println("a * b = " + (a * b) );
        System.out.println("b / a = " + (b / a) );
        System.out.println("b % a = " + (b % a) );
        System.out.println("b % a = " + (b % a) );
        System.out.println("b++   = " +  (++b) );
        System.out.println("b--   = " + (--b) );

    }
}
```

```
Output:

a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
b % a = 0
b++   = 21
b--   = 20
```

The Relational Operators

There are following relational operators supported by Java language.
Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Example:

```java
public class App {
    public static void main(String[] args) throws Exception {
    int a = 10;
    int b = 20;
    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
    System.out.println("b >= a = " + (b >= a) );
    System.out.println("b <= a = " + (b <= a) );
    }
```

```
}
```

Output:
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false

The Bitwise Operators
Java defines several bitwise operators, which can be applied to the i**nteger types, long, int, short, char, and byte.**
Bitwise operator works on **bits** and performs **bit-by-bit** operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –
To Know Binary Concept:195-64

Value	**128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**
64	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**
13	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**1**

a = 0011 1100
b = 0000 1101
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

a	**b**	&(AND)	I(OR)	^(XOR)	~NOT(Complement)
0	**0**	0	0	Similar 0	1(Opppsite)
0	**0**	0	0	0	1
1	**0**	0	1	1	0
1	**0**	0	1	1	0
1	**1**	1	1	0	0
1	**1**	1	1	0	0
0	**0**	0	0	0	1
0	**1**	0	1	Different 1	1
Result		12	61	49	-61

The following table lists the bitwise operators –
Assume integer variable A holds 60 and variable B holds 13 then –
0 for false and 1 for True

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit	(A & B) will give 12 which is 0000 1100

	to the result if it exists in both operands.	
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift) Unsigned shift right	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Example:

```
 int a = 60; /* 60 = 0011 1100 */

        int b = 13; /* 13 = 0000 1101 */
        int c = 0;

        c = a & b;        /* 12 = 0000 1100 */
        System.out.println("a & b = " + c );

        c = a | b;        /* 61 = 0011 1101 */
        System.out.println("a | b = " + c );
```

```
Output:
a & b = 12
a | b = 61
```

Shift Operator Calculation:

<< left shift

A << 2

Now Add two zero in last

~~00~~11 110000=240

>> rt shift

A>>2

0011 11~~00~~=15

>>> Shift
A>>>2
00001111~~00~~=15

The Logical Operators:
The following table lists the logical operators –
Assume Boolean variables A holds true and variable B holds false, then –

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Example:

```
public class App {
    public static void main(String[] args) throws Exception {
        boolean a = true;
        boolean b = false;
        System.out.println("a && b = " + (a&&b));
        System.out.println("a || b = " + (a||b) );
        System.out.println("!(a && b) = " + !(a && b));

    }
}
```

Output:
a && b = false
a || b = true
!(a && b) = true

The Assignment Operators
Following are the assignment operators supported by Java language –

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C − A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Example:

```
public class App {
    public static void main(String[] args) throws Exception {
        int a = 10;
        int b = 20;
        int c = 0;
        c = a + b;
        System.out.println("c = a + b = " + c );
        c += a ;
        System.out.println("c += a  = " + c );
        c -= a ;
        System.out.println("c -= a = " + c );
        c *= a ;
        System.out.println("c *= a = " + c );

    }
}
```

Output:
c = a + b = 30
c += a = 40
c -= a = 30
c *= a = 300

Example:

```java
public class App {
    public static void main(String[] args) throws Exception {
        int a = 10;
      int c = 15;

      c /= a ;//15/10=1
      System.out.println("c /= a = " + c );

      c = 15;
      c %= a ;//15%10=remainder=5
      System.out.println("c %= a  = " + c );

      c = 15;
      c &= a ;
      Explanation:
      C=c&a
      1 1 1 1
      1 0 1 0
    = 1 0 1 0=10
      System.out.println("c &= a  = " + c );

      c = 15;
      c ^= a ;//

      Explanation:
      C=c^a
      1 1 1 1
      1 0 1 0
    = 0 1 0 1=5(If similar than 0 if not similar than 1)

      System.out.println("c ^= a   = " + c );

      c = 15;
      c |= a ;
      Explanation:
      C=c|a
      1 1 1 1
      1 0 1 0
    = 1 1 1 1=15
      System.out.println("c |= a   = " + c );

    }
}
```

Output:

c /= a = 1
c %= a = 5
c &= a = 10
c ^= a = 5
c |= a = 15

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of **three** operands and is used to evaluate **Boolean expressions**. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

variable x = (expression) ? value if true : value if false

Example
In this example, we're creating two variables a and b and using ternary operator we've decided the values of **b** and printed it.

```
int a, b;
a = 10;
b = (a == 1) ? 20: 30;
System.out.println( "Value of b is : " +  b );//30

b = (a == 10) ? 20: 30;
System.out.println( "Value of b is : " + b );//20
```

Output:
Value of b is : 30
Value of b is : 20

Control Statement:
Control Statements are the base of any programming language. Using control statements we implement real world scenarios in programs. Java provides statements that can be used to control the flow of Java code. Such statements are called **control flow statements**. There are three types of Control Statements in Java:
Decision Making statements

- if statements
- if..else statement
- nested if..else statement
- switch statement

Loop statements

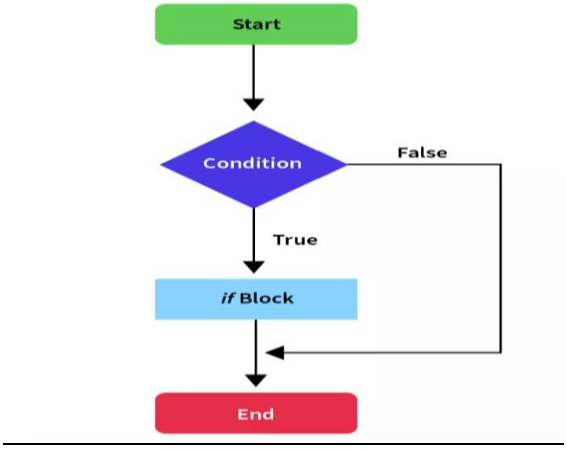
- do while loop
- while loop
- for loop
- for-each loop

Jump statements

- break statement
- continue statement
- return

Decision making statements execute a piece of code based on **some condition**.
Looping Statements execute a piece of code **repeatedly** until a condition becomes false.
Jump or branch statements help in **transferring the control** of the flow of execution to a **specific point** in the code.

<u>If Statement</u>
If the condition is true, then the code is executed otherwise not.



In the above flow diagram, we can see that whenever the condition is **true**, we execute the **if** block otherwise we skip it and continue the execution with the code following the **if** block.

Syntax:

```
if(condition) {

  // block of code to be executed if the condition is true
}
```

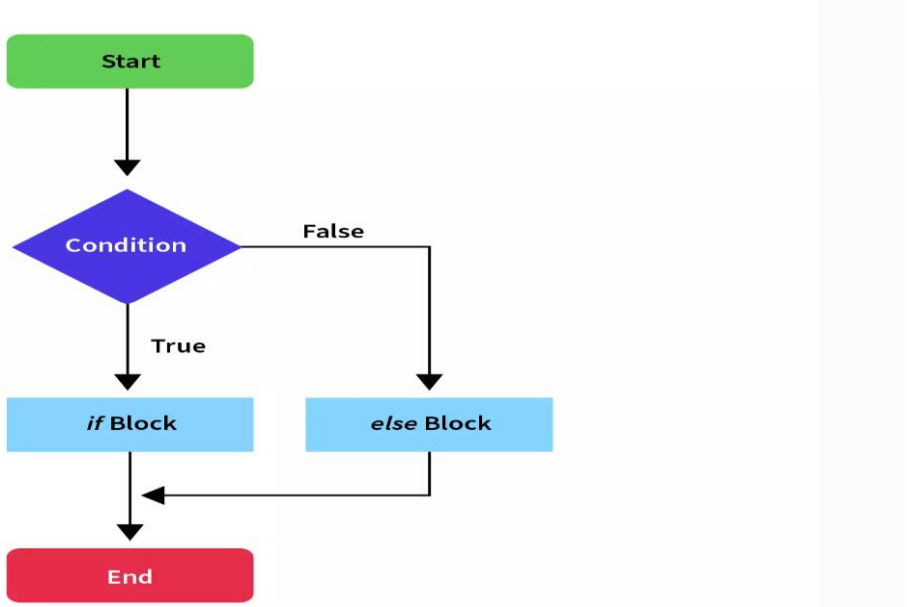
Example:

```
String role = "admin";

if(role == "admin") {
    System.out.println("Admin screen is loaded");
}
```

<u>if..Else Statement:</u>

if the condition is **true** then the code inside the **if** block is executed otherwise the **else** block is executed.



The above flow diagram is similar to the **if statement**, with a difference that whenever the condition is **false**, we execute the **else** block and then continue the normal flow of execution.

Syntax:

```
if (condition) {
  // If block executed when the condition is true
}
else {
      // Else block executed when the condition is false
}
```

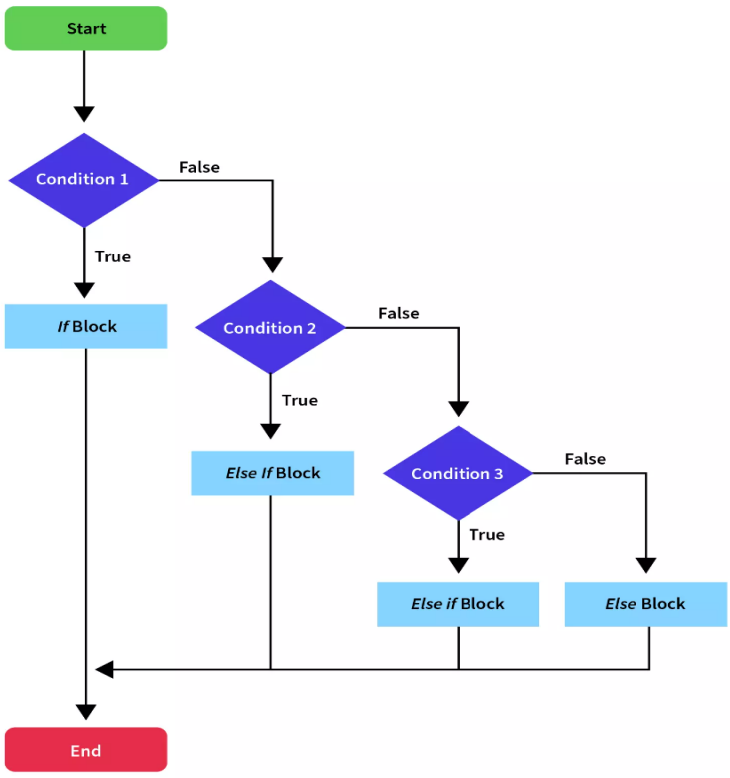
Example:

```
String role = "user";
if(role == "admin") {
    System.out.println("Admin screen is loaded");
}
else {
    System.out.println("User screen is loaded");
}
```

<u>if-else-if Ladder:</u>

if statement is followed by **multiple else-if** blocks. We can create a decision tree by using these control statements in Java in which the block where the condition is **true** is executed and the rest of the ladder is ignored and not executed.

If none of the conditions is **true**, the last **else** block is executed,



As you can see in the above flow diagram of the if-else ladder, we execute the if block if the condition is true, otherwise if the condition is false, instead of executing the else block, we check other multiple conditions to determine which block of code to execute.
If none of the conditions are true, the last else block, if present, is executed.

Syntax:

```
if(condition1) {
    // Executed only when the condition1 is true
}
else if(condition2) {
    // Executed only when the condition2 is true
}
.
.
else {
    // Executed when all the conditions mentioned above are true
}
```

Example:

```
String browser = "chrome";
if(browser == "safari") {
    System.out.println("The browser is safari");
}
else if(browser == "edge") {
    System.out.println("The browser is edge");
}
else if(browser == "chrome"){
    System.out.println("The browser is chrome");
}
else {
    System.out.println("Not a supported browser");
}
```

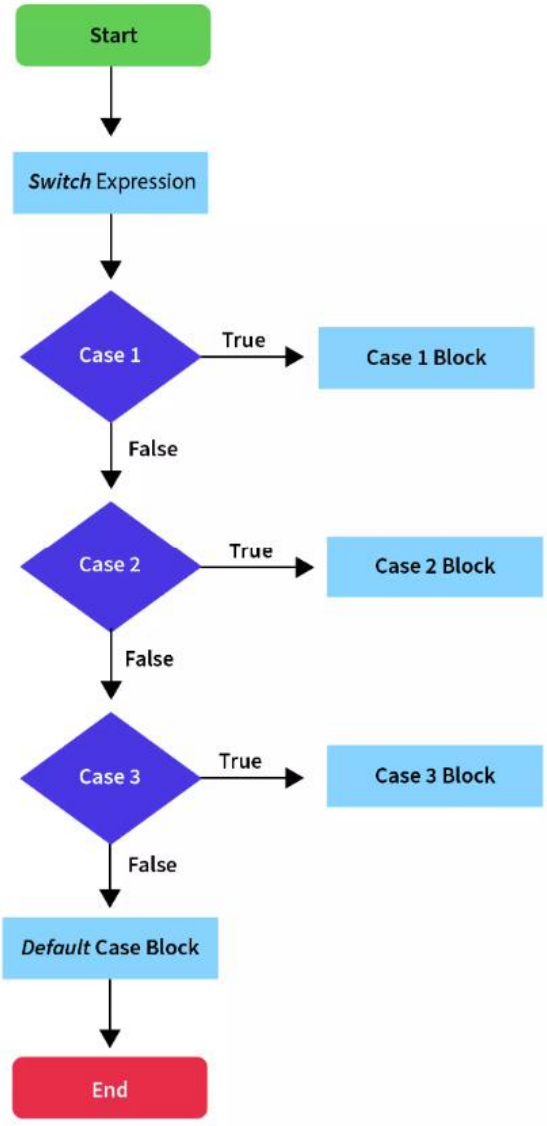
switch Statement

Switch statements are almost similar to the **if-else-if ladder** control statements in Java. It is a multi-branch statement. It is a bit easier than the **if-else-if ladder** and also more **user-friendly and readable**. The **switch** statements have an **expression** and based on the output of the expression, one or more blocks of codes are executed.

Syntax:

```
switch (expression) {
  case value1:
      //code block of case with value1
      break;
  case value2:
      //code block of case with value2
      break;
```

```
  .
  .
  case valueN:
      //code block of case with valueN
      break;
  default:
     //code block of default value
}
```



In the above flow diagram, we have a switch expression and we match the output of the expression through a series of case blocks.
Whichever case matches the output, its block is executed and execution skips to the end of the switch; otherwise, if none of the cases matches, the default block is executed.
Here, we have multiple case statements and each case code block is followed by a break statement to stop the execution to that case only.
Example:

```
String browser = "chrome";
switch (browser)
{
    case "safari":
        System.out.println("The browser is Safari");
        break;
    case "edge":
        System.out.println("The browser is Edge");
        break;
    case "chrome":
        System.out.println("The browser is Chrome");
        break;
    default:
        System.out.println("The browser is not supported");
}
```

Loop Statements

Java provides a set of **looping** statements that **executes a block of code repeatedly** while some condition evaluates to true. Looping control statements in Java are used to traverse a collection of elements, like arrays.

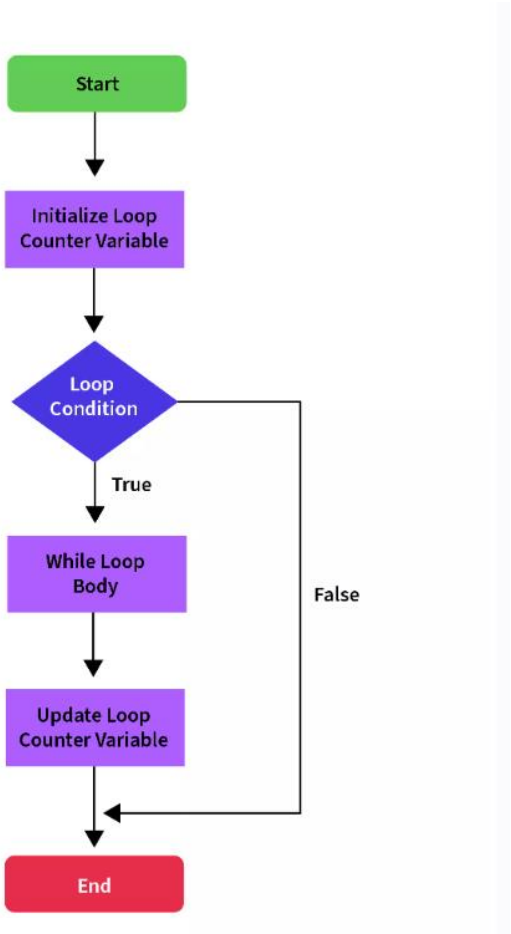
While Loop

The **while loop** statement is the simplest kind of loop statement. It is used to iterate over a single statement or a block of statements until the specified boolean condition is **false**.
The while loop statement is also called the **entry-control looping statement**
Syntax:

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

Example:

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



do-while Loop

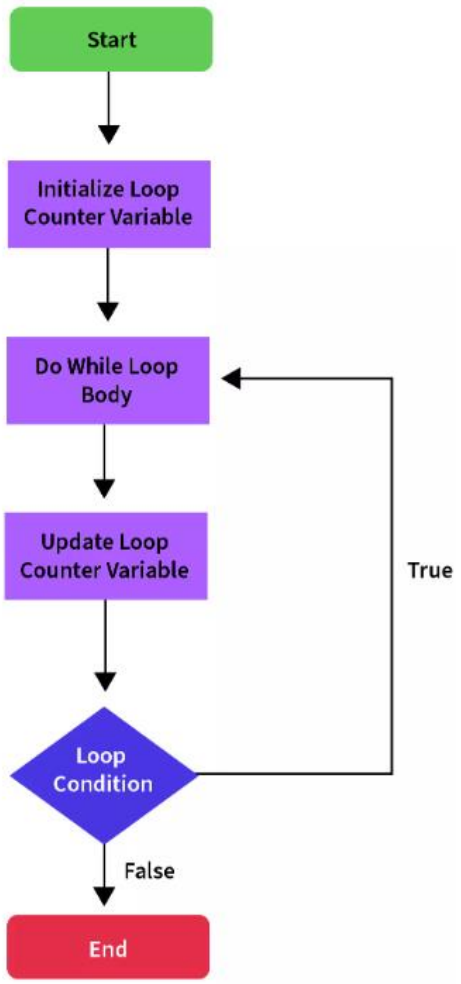
The Java **do-while loop** statement works the same as the **while loop** statement with the only difference being that its boolean condition is evaluated **post first execution of the body** of the loop. Thus it is also called **exit controlled looping statement**.

Syntax:

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

Example:

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



for Loop

In a **for loop statement**, execution begins with the **initialization** of the looping variable, then it executes the **condition**, and then it **increments** or **decrements** the looping variable.

If the condition results in **true** then the loop body is executed otherwise the for loop statement is **terminated**.

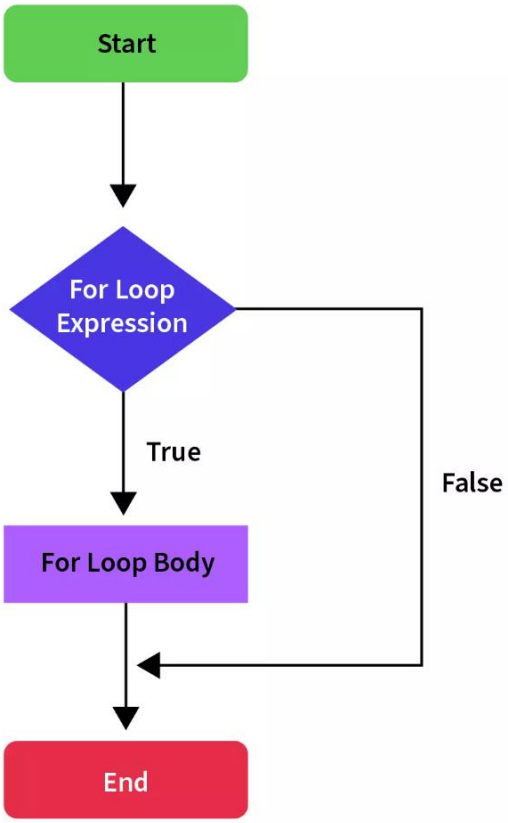
Syntax:

```
for (initialization; condition; increment/decrement)
{
    // code block to be executed if condition is true
}
```

Example:

```
for (int i = 0; i < 5; i++)

{
  System.out.println(i);
}
```



for-each Loop

The **for-each loop** statement provides an approach to traverse through elements of an array or a collection in Java. It executes the body of the loop for each element of the given array or collection. It is also known as the **Enhanced for loop statement** because it is easier to use than the for loop statement as you don't have to handle the **increment operation**.

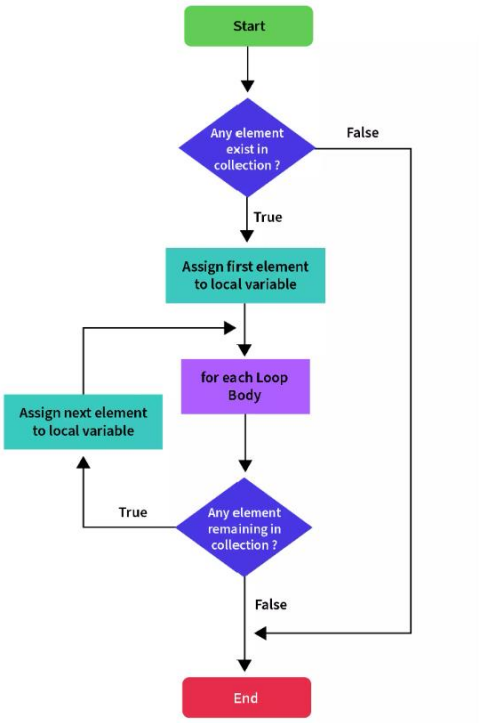
Syntax:

```
for(dataType variableName : array | collection)

{
    // code block to be executed
}
```

Example:

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



Jump/Branching Statements

Jump/Branching control statements in Java **transfer the control** of the program to **other blocks** or parts of the program and hence are known as **the branch or jump** statements.

Break Statement

The break statement as we can deduce from the name is used to **break** the current flow of the program. The **break** statement is commonly used in the following three situations:

- Terminate a case block in **a switch statement** as we saw in the example of the switch statement in the above section.
- To **exit** the loop explicitly, even if the loop condition is **true**.
- Use as the alternative for the **goto** statement along with java labels, since java doesn't have **goto** statements.

The **break** statement cannot be used as a **standalone** statement in Java. It must be either inside **a switch or a loop.** If we try to use it outside a loop or a switch, JVM will give an error.

Syntax:

```
for(condition) {
    // body of the loop
    break;
}
while(condition) {
    // body of the loop
    break;
}
```

Example:

```
for(int i = 0; i < 10; i++) {
  System.out.println("The value of the index is: " + index);
  if(i == 3) {
      break;
  }
```

Output:

The value of the index is: 0

The value of the index is: 1

The value of the index is: 2

The value of the index is: 3

<u>Continue Statement</u>

Sometimes there are situations where we just want to ignore the rest of the code in the loop body and continue from the next iteration. The **continue** statement in Java allows us to do just that. This is similar to the **break** statement in the sense that it **bypasses** every line in the loop body after itself, but instead of exiting the loop, it goes to the next iteration.

Syntax:

```
for(condition) {
  // body of the loop
  continue;
  //the statements after this won't be executed
}
```

Another Syntax:

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

```
        continue;
        // the statements after this won't be executed
    }
```

Example:

```
      System.out.println("The odd numbers between 1 to 10 are: ");
      for (int number = 1; number <= 10; number++) {
          if (number % 2 == 0) continue;
          System.out.println(number);
      }
```

Output:

The odd numbers between 1 to 10 are:

1

3

5

7

9

Return Statement

The **return** statements are used when we need to return from a method explicitly. The return statement transfers the control back to the caller method of the current method. In the case of the main method, the execution is completed and the program is terminated. Return statements are often used for conditional termination of a method or to return something from the method to the caller method.

Syntax:

```
      void method() {
        // body of the method
        return;
    }
```

Example:

```
public class App {
    public static void main(String[] args) throws Exception {
      System.out.println(search(3));
      System.out.println(search(10));

    }
    public static String search(int key) {
```

```
        int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        for (int element : numbers) {
          if (element == key) {
            return "Success";
            // Putting statements post this return statement
            // Will throw compile-time error
          }
        }
        return "Failure";
    }
}
```

Output:

Success

Failure