Unit II

Constants, Variables and Data Types: Constants ,Variables ,Data Types, Declaration of variables, Giving values to variables, Symbolic Constants, Typecasting.

Operators & Expression: Arithmetic operators, Relational operators, Logical operators, Assignment operators, Increment & Decrement operators, Conditional operators, Bitwise operators, Arithmetic Expressions, Evaluation of Expression, Operator Presentence & Associativity.

Decision Making, Branching & Looping: Decision Making with control statements, Looping statements, Jump in Loop

Data Types in Java

Java is statically typed and also a strongly typed language because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the Java data types.

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

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

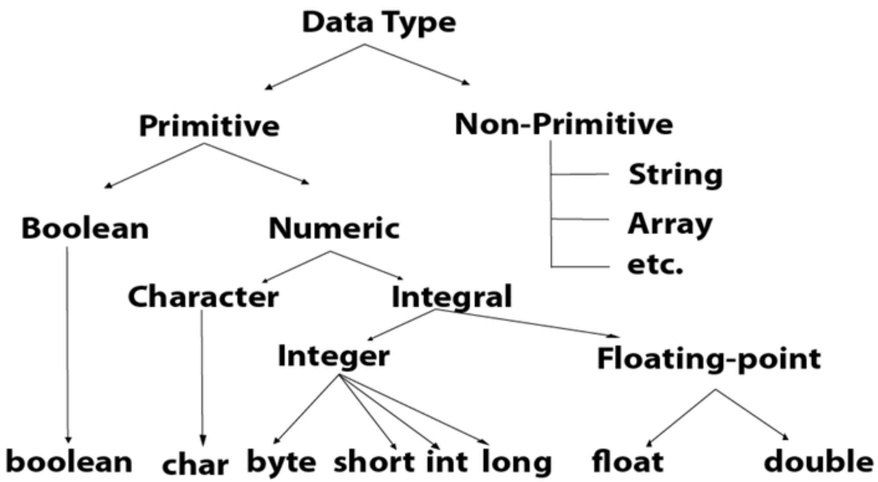
Java Primitive Data Types

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

In Java, there are mainly eight primitive data types and let's understand about them in detail.

Java 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

Primitive

Non-Primitive

String

Array

etc.

Boolean

Numeric

Character

Integral

Integer

Floating-point

boolean

char

byte

short

int

long

float

double

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

In Java, the **boolean** data type represents a single bit of information with two possible states: **true** or **false**. It is used to store the result of logical expressions or conditions. Unlike other primitive data types like **int** or **double**, **boolean** does not have a specific size or range. It is typically implemented as a single bit, although the exact implementation may vary across platforms.

Example

```
boolean a=false;
boolean b=true;
System.out.println("a= " + a);
System.out.println("b= " + b);
```

Byte Data Type

The byte data type in Java is a primitive data type that represents an 8-bit signed two's complement integer. It has a range of values from -128 to 127. Its default value is 0. The byte data type is commonly used when working with raw binary data or when memory conservation is a concern, as it occupies less memory than larger integer types like int or long.

Example

```
byte a=10;
byte b=-20;
System.out.println("a= " + a);
System.out.println("b= " + b);
```

Short Data Type

The **short** data type in Java is a primitive data type that represents a 16-bit signed two's complement integer. It has a range of values from -32,768 to 32,767. Similar to the **byte** data type, **short** is used when memory conservation is a concern, but more precision than **byte** is required. Its default value is 0.

Example

```
short a=10000;
short b=-5000;
System.out.println("a= " + a);
System.out.println("b= " + b);
```

Int Data Type

The int data type in Java is a primitive data type that represents a 32-bit signed two's complement integer. It has a range of values from -2,147,483,648 to 2,147,483,647. The int data type is one of the most commonly used data types in Java and is typically used to store whole numbers without decimal points. Its default value is 0.

Example

```
int a=100000;
int b=-200000;
System.out.println("a= " + a);
System.out.println("b= " + b);
```

Long Data Type

The **long** data type in Java is a primitive data type that represents a 64-bit signed two's complement integer. It has a wider range of values than **int**, ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Its default value is 0.0F. The **long** data type is used when **int** is not large enough to hold the desired value, or when a larger range of integer values is needed.

Example

```
long a = 5000000L;
long b = -6000000L;
System.out.println("a= " + a);
System.out.println("b= " + b);
```

Float Data Type

The float data type in Java is a primitive data type that represents single-precision 32-bit IEEE 754 floating-point numbers. It can represent a wide range of decimal values, but it is not suitable for precise values such as currency. The float data type is useful for applications where a higher range of values is needed, and precision is not critical.

Example

```
float f = 234.5f;
System.out.println("f = " + f);
```

Double Data Type

The double data type in Java is a primitive data type that represents double-precision 64-bit IEEE 754 floating-point numbers. Its default value is 0.0d. It provides a wider range of values and greater precision compared to the float data type, making it suitable for applications where accurate representation of decimal values is required.

Example

```
double d = 12.3;
System.out.println("d = " + d);
```

Char Data Type

The **char** data type in Java is a primitive data type that represents a single 16-bit Unicode character. It can store any character from the Unicode character set, that allows Java to support internationalization and representation of characters from various languages and writing systems.

Example

```
char c = 'A';
System.out.println("c = " + c);
```

Non-Primitive Data Types in Java

In Java, non-primitive data types, also known as reference data types, are used to store complex objects rather than simple values. Unlike primitive data types that store the actual values, reference data types store references or memory addresses that point to the location of the object in memory. This distinction is important because it affects how these data types are stored, passed, and manipulated in Java programs.

Class

One common non-primitive data type in Java is the class. Classes are used to create objects, which are instances of the class. A class defines the properties and behaviors of objects, including variables (fields) and methods. For example, you might create a **Person** class to represent a person, with variables for the person's name, age, and address, and methods to set and get these values.

Interface

Interfaces are another important non-primitive data type in Java. An interface defines a contract for what a class implementing the interface must provide, without specifying how it should be implemented. Interfaces are used to achieve abstraction and multiple inheritance in Java, allowing classes to be more flexible and reusable.

Arrays

Arrays are a fundamental non-primitive data type in Java that allow you to store multiple values of the same type in a single variable. Arrays have a fixed size, which is specified when the array is created, and can be accessed using an index. Arrays are commonly used to store lists of values or to represent matrices and other multi-dimensional data structures.

Enum

Java also includes other non-primitive data types, such as enums and collections. Enums are used to define a set of named constants, providing a way to represent a fixed set of values. Collections are a framework of classes and interfaces that provide dynamic data structures such as lists, sets, and maps, which can grow or shrink in size as needed.

Overall, non-primitive data types in Java are essential for creating complex and flexible programs. They allow you to create and manipulate objects, define relationships between objects, and represent complex data structures. By understanding how to use non-primitive data types effectively, you can write more efficient and maintainable Java code.

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: local, instance and static.

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

int data=50;*//Here data is variable*

How to Declare Java Variables?

We can declare variables in Java as pictorially depicted below:

datatype: In Java, a data type define the type of data that a variable can hold.

data_name: Name was given to the variable.

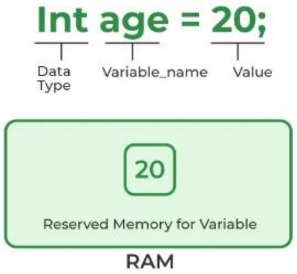
In this way, a name can only be given to a memory location. It can be assigned values in two ways:

Variable Initialization

Assigning value by taking input

How to Initialize Java Variables?

It can be perceived with the help of 3 components explained above:



Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

1)Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

Example

```
//defining a Local Variable
int num = 10;
System.out.println(" Variable: " + num);
```

Output:

```
Variable: 10
```

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

```
public class InstanceVariableDemo {
    //Defining Instance Variables
    public String name;
    public int age=19;
 //Creadting a default Constructor initializing Instance Variable
    public InstanceVariableDemo()
    {
        this.name = "Deepak";
    }
}
public class Main{
    public static void main(String[] args)
    {
        // Object Creation
       InstanceVariableDemo obj = new InstanceVariableDemo();
        System.out.println("Student Name is: " + obj.name);
        System.out.println("Age: "+ obj.age);
    }
}
```

Output:

```
Student Name is: Deepak
Age: 19
```

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.

```
class Student{
   //static variable
  static int age;
}
public class Main{
  public static void main(String args[]){
     Student s1 = new Student();
     Student s2 = new Student();
     s1.age = 24;
     s2.age = 21;
     Student.age = 23;
     System.out.println("S1\'s age is: " + s1.age);
     System.out.println("S2\'s age is: " + s2.age);
  }
}
```

Output:

```
S1's age is: 23
S2's age is: 23
```

Constant

A **constant** is an entity in programming that is immutable. In other words, the value that cannot be changed. Usually, to accomplish this, the variable is declared using the final keyword. Constants are frequently used to represent stable values, like mathematical constants, configuration settings, or flag values, that do not change while a program is running. A variable's value is guaranteed to stay constant and unintentionally changed if it is declared as a constant.

Constant is a value that cannot be changed after assigning it. 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.

How to declare constant in Java?

In Java, to declare any variable as constant, we use static and final modifiers. It is also known as **non-access** modifiers. According to the Java naming convention the identifier name must be in **capital letters**.

Static and Final Modifiers

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;
```

For example, price is a variable that we want to make constant.

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

Where static and final are the non-access modifiers. The double is the data type and PRICE is the identifier name in which the value 432.78 is assigned.

In the above statement, the **static** modifier causes the variable to be available without an instance of its defining class being loaded and the **final** modifier makes the variable fixed.

Here a question arises that **why we use both static and final modifiers to declare a constant?**

If we declare a variable as **static**, all the objects of the class (in which constant is defined) will be able to access the variable and can be changed its value. To overcome this problem, we use the **final** modifier with a static modifier.

When the variable defined as **final**, the multiple instances of the same constant value will be created for every different object which is not desirable.

When we use **static** and **final** modifiers together, the variable remains static and can be initialized once. Therefore, to declare a variable as constant, we use both static and final modifiers. It shares a common memory location for all objects of its containing class.

Why we use constants?

The use of constants in programming makes the program easy and understandable which can be easily understood by others. It also affects the performance because a constant variable is cached by both JVM and the application.

Points to Remember:

- Write the identifier name in capital letters that we want to declare as constant. For example, MAX=12.
- If we use the private access-specifier before the constant name, the value of the constant cannot be changed in that particular class.
- If we use the public access-specifier before the constant name, the value of the constant can be changed in the program.

Tokens

The Java compiler breaks the line of code into text (words) is called **Java tokens**. These are the smallest element of the Java program. The Java compiler identified these words as tokens. These tokens are separated by the delimiters. It is useful for compilers to detect errors. Remember that the delimiters are not part of the Java tokens.

token <= identifier | keyword | operator | comment

For example, consider the following code.

```
 public class Demo
{
public static void main(String args[])
{
System.out.println("javatpoint");
}
}
```

In the above code snippet, **public, class, Demo, {, static, void, main, (, String, args, [,],), System, ., out, println, javatpoint**, etc. are the Java tokens.

The Java compiler translates these tokens into Java bytecode. Further, these bytecodes are executed inside the interpreted Java environment.

Types of Tokens

Java token includes the following:

- Keywords
- Identifiers
- Operators
- Comments

Keywords: These are the **pre-defined** reserved words of any programming language. Each keyword has a special meaning. It is always written in lower case. Since keywords are referred names for a compiler, they can’t be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. Java provides the following keywords:

01. abstract	02. boolean	03. byte	04. break	05. class
06. case	07. catch	08. char	09. continue	10. default
11. do	12. double	13. else	14. extends	15. final
16. finally	17. float	18. for	19. if	20. implements
21. import	22. instanceof	23. int	24. interface	25. long
26. native	27. new	28. package	29. private	30. protected
31. public	32. return	33. short	34. static	35. super
36. switch	37. synchronized	38. this	39. thro	40. throws
41. transient	42. try	43. void	44. volatile	45. while
46. assert	47. const	48. enum	49. goto	50. strictfp

2.Identifiers

Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user-defined names consisting of an arbitrarily long sequence of letters and digits with either a letter or the underscore (_) as a first character. Identifier names must differ in spelling and case from any keywords. You cannot use keywords as identifiers; they are reserved for special use. Once declared, you can use the identifier in later program statements to refer to the associated value. A special kind of identifier, called a statement label, can be used in goto statements

There are some rules to declare identifiers are:

- ❑ The first letter of an identifier must be a letter, underscore or a dollar sign. It cannot start with digits but may contain digits.
- ❑ The whitespace cannot be included in the identifier.
- ❑ Identifiers are case sensitive.

Some valid identifiers are:

PhoneNumber
PRICE
radius
a
a1
_phonenumber
$circumference
jagged_array
12radius *//invalid*

3. Operators

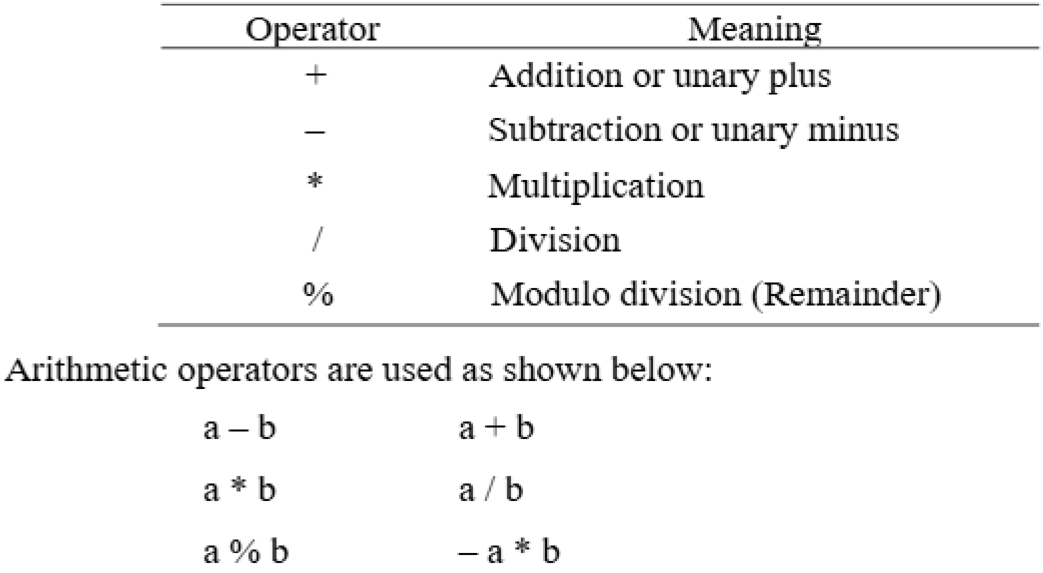
Java supports a rich set of operators. We have already used several of them, such as =, +, –, and *. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of mathematical or logical expressions

Java operators can be classified into a number of related categories as below:

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators

1.Arithmetic operators

Arithmetic operators are used to construct mathematical expressions as in algebra. Java provides all the basic arithmetic operators. The operators +, –, *, and / all works the same way as they do in other languages. These can on any built-in numeric data type of Java. We cannot use these operators on Boolean type. The unary minus operator, in effect, multiplies its single operand by –1. Therefore, a number preceded by a minus sign changes its sign.



Operator	Meaning
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division (Remainder)

Arithmetic operators are used as shown below:

a – b   a + b

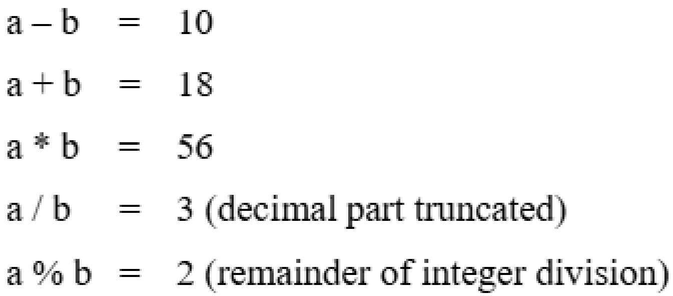
a * b   a / b

a % b   – a * b

Here a and b may be variables or constants and are known as operands.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a + b are integers, the expression is called an integer expression, and the operation is called integer arithmetic. Integer arithmetic always yields an integer value. In the above examples, if a and b are integers, then for a = 14 and b = 4 we have the following results:



a – b = 10

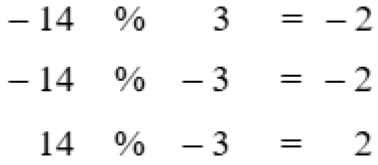
a + b = 18

a * b = 56

a / b = 3 (decimal part truncated)

a % b = 2 (remainder of integer division)

a/b, when a and b are integer types, gives the result of division of a by b after truncating the divisor. This operation is called the integer division.
For modulo division, the sign of the result is always the sign of the first operand (the dividend). That is



– 14 % 3 = – 2

– 14 % – 3 = – 2

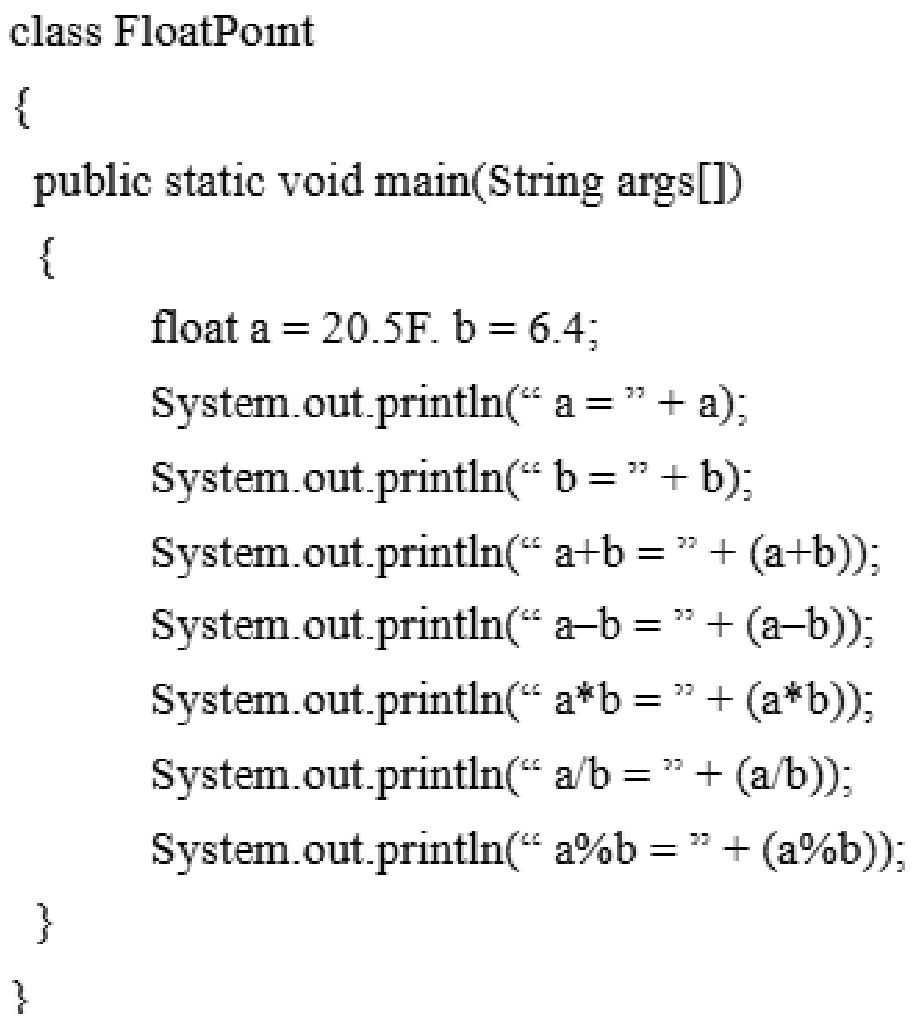
14 % – 3 = 2

(Note that module division is defined as : a%b = a – (a/b)*b, where a/b is the integer division).

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

Unlike C and C++, modulus operator % can be applied to the floating point data as well. The floating point modulus operator returns the floating point equivalent of an integer division. What this means is that the division is carried out with both floating point operands, but the resulting divisor is treated as an integer, resulting in a floating point remainder. Program



```
class FloatPoint
{
 public static void main(String args[])
  {
           float a = 20.5F. b = 6.4;
           System.out.println(“ a = ” + a);
           System.out.println(“ b = ” + 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(“ a/b = ” + (a/b));
           System.out.println(“ a%b = ” + (a%b));
  }
}
```

The output of Program is as follows:

```
a = 20.5
b = 6.4
a+b = 26.9
a–b = 14.1
a*b = 131.2
a/b = 3.20313
a%b = 1.3
```

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then the other operand is converted to real and the ral arithmetic is performed. The result will be a real. Thus

15/10.0 produces the result 1.5

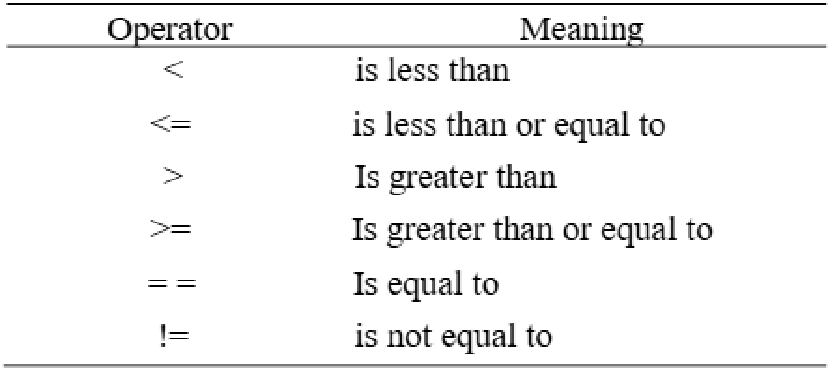
Whereas

15/10 produces the result 1

2. Relational Operators

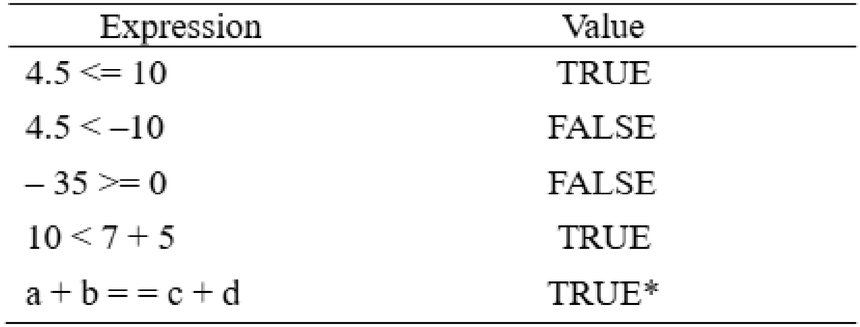
We often compare two quantities, and depending on their relation, take certain
decisions. Relational expressions are used in decision statements such as, if and while to decide the course of action of a running program. For example, we may compare the age of two persons, or the price of two items, and
so on. These comparisons can be done with the help of relational operators. We have already
used the symbol '<' meaning 'less than'. An expression such as
a < b or x < 20
containing a relational operator is termed as a relational expression. The value of relational
expression is either true or false. For example, if x = 10, then
x < 20 is true
while
20 < x is false.
Java supports six r anings are

Relational Operat



Operator	Meaning
<	is less than
<=	is less than or equal to
>	Is greater than
>=	Is greater than or equal to
= =	Is equal to
!=	is not equal to

Expression



Expression	Value
4.5 <= 10	TRUE
4.5 < –10	FALSE
– 35 >= 0	FALSE
10 < 7 + 5	TRUE
a + b = = c + d	TRUE*

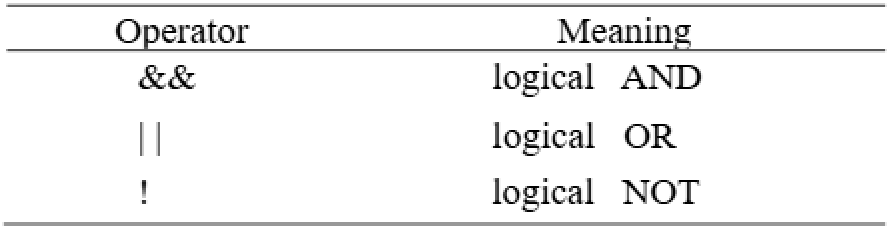
```
class RelationalOperators
{
public static void main(String args[])
{
float a = 15.0F, b = 20.75F, c = 15.0F;
System.out.println(“ a = ” + a);
System.out.println(“ b = ” + b);
System.out.println(“ c = ” + c);
System.out.println(“ a < b is ” + (a<b));
System.out.println(“ a > b is ” + (a>b));
System.out.println(“ a == c is ” + (a==c));
System.out.println(“ a <= c is ” + (a<=c));
System.out.println(“ a >= b is ” + (a>=b));
System.out.println(“ b != c is ” + (b!=c));
System.out.println(“ b == a+c is ” + (b==a+c));
}
}
```

Otuput

```
a = 15
b = 20.75
c = 15
a < b is true
a > b is false
a == c is true
a <= c is true
a >= b is false
a != c is true
b == a+c is false
```

3. Logical Operators

In addition to the relational operators, Java has three logical operators, which are given in



Operator	Meaning
&&	logical AND
| |	logical OR
!	logical NOT

The logical operators && and | | are used when we want to form compound conditions by combining two or more relations. An example is:

a > b && x == 10

An expression of this kind which combines two or more relational expressions is termed as a logical expression or a compound relational expression.

Note:

□ op – 1 && op – 2 is true if both op – 1 and op – 2 are true and false otherwise.

□ op – 1 | | op – 2 is false if both op – 1 and op – 2 are false and true otherwise.

Some examples of the usage of logical expression are:

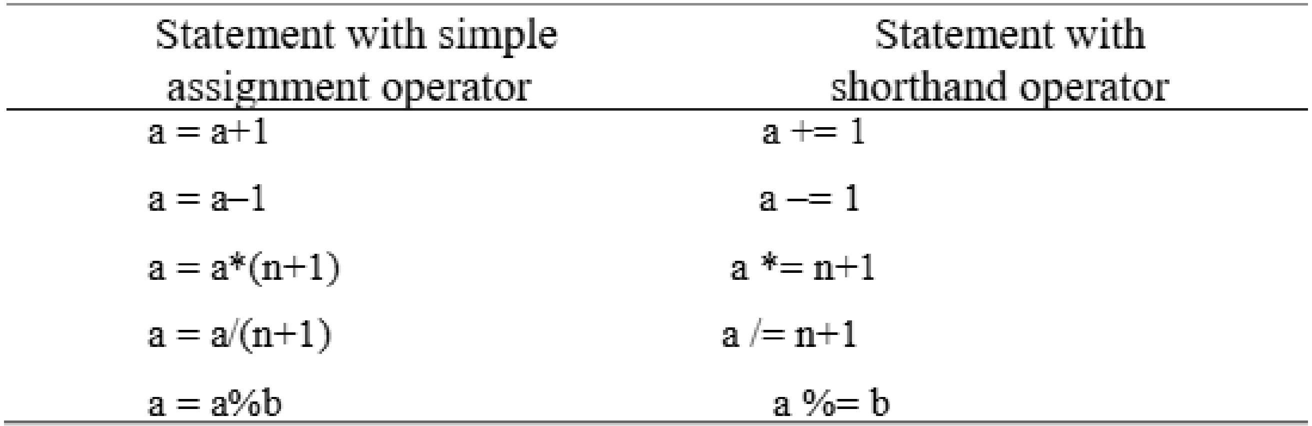
1. if (age>55 && salary<1000)
2. if (number<0) || number>1000)

4. Assignment Operators

Assignment operators are used to assign the value of an expression to a variable. We have seen the usual assignment operator, ‘=’. In addition, Java has a set of ‘shorthand’ assignment operators which are used in the form

v op= exp;

where v is a variable, exp is an expression and op is a Java binary operatory. The operator op = is known as the shorthand assignment operator.



Statement with simple assignment operator	Statement with shorthand operator
a = a+1	a += 1
a = a–1	a –= 1
a = a*(n+1)	a *= n+1
a = a/(n+1)	a /= n+1
a = a%b	a %= b

5. Increment and Decrement Operators

Java has two very useful operators not generally found in many other languages. These are the increment and decrement operators.

++ and —

The operator ++ adds 1 to the operand while — subtracts 1. Both are unary operators and are used in the following form:

++m; or m++;

—m; or m—;

++m; is equivalent to m = m + 1; (or m += 1;)

—m; is equivalent to m = m – 1; (or m –= 1;)

We use the increment and decrement operators extensively in for and while loops.

While ++m and m++ mean the same thing when they from statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

m = 5;

y = ++m;

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as

m = 5;

y = m++;

then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand

```
class IncrementOperator
{
public static void main(String args[])
{
int m = 10, n = 20
System.out.println(“ m = ” + m);
System.out.println(“ n = ” + n);
System.out.println(“ ++m = ” +++m n);
System.out.println(“ n++ = ” + n++);
System.out.println(“ m = ” + m);
System.out.println(“ n = ” + n);
}
}
```

Output

```
m = 10
n = 20
++m = 11
n++ = 20
m = 11
n = 21
```

6. Conditional Operator

The character pair ? : is a ternary operatory available in Java. This operator is used to construct conditional expressions of the form

exp1 ? exp2 : exp3

where exp1, exp2, and exp3 are expressions.

The operator ? : works as follows: exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the conditional expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the conditional expression. None that only one of the expressions (either exp2 or exp3) is evaluated. For example, consider the following statements:

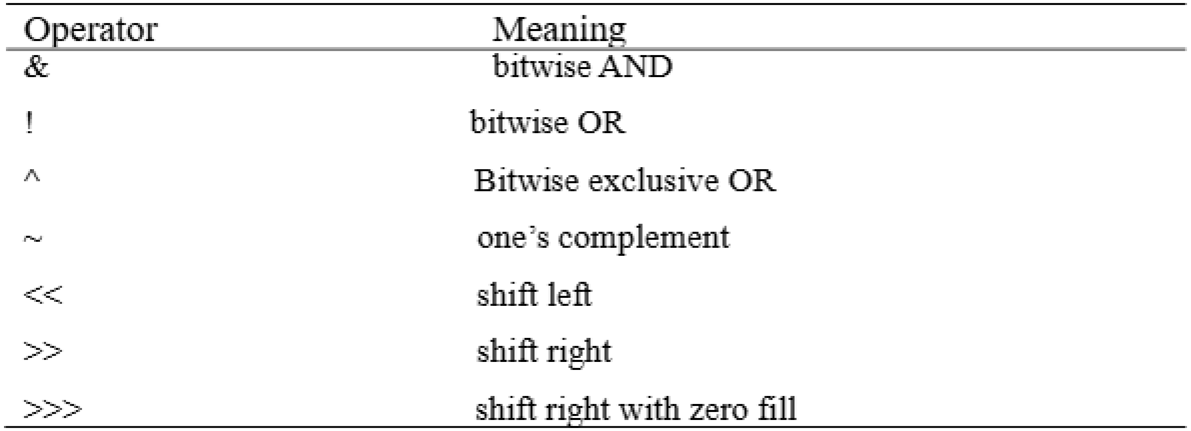
```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the if….else statement as follows:

```
if(a > b)
x = a;
else
x = b;
```

7. Bitwise Operators

Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to float or double.



Operator	Meaning
&	bitwise AND
!	bitwise OR
^	Bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right
>>>	shift right with zero fill

4. Comments

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

Why do we use comments in a code?

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.

Types of Java Comments

There are three types of comments in Java.

1) Single Line Comment
2) Multi Line Comment
3) Documentation Comment

1) Java 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.

Syntax:

```
//This is single line comment
```

Example

```
public class CommentExample1 {
public static void main(String[] args) {
int i=10; // i is a variable with value 10
System.out.println(i);  //printing the variable i
}
}
```

Output:

```
10
```

2) Java 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.

Syntax:

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

Example

```
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
  int i=10;
    System.out.println(i);
/* float j = 5.9;
    float k = 4.4;
    System.out.println( j + k ); */
}
}
```

Output:

```
10
```

3) Java Documentation Comment

Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API. These APIs are needed for reference, i.e., which classes, methods, arguments, etc., are used in the code.

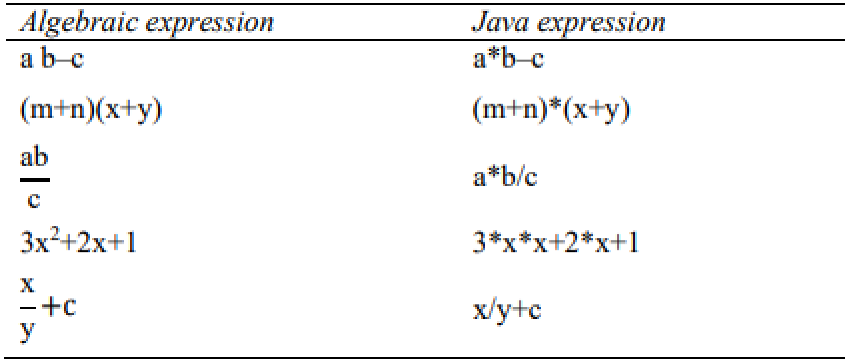
To create documentation API, we need to use the **javadoc tool**. The documentation comments are placed between /** and */.

Syntax:

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

Arithmetic Expressions

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. We have used a number of simple expression in the examples discussed so far. Java can handle any complex mathematical expressions. Some of the examples of Java expressions are shown in Table shows that Java does not have an operator for exponentiation.



Algebraic expression	*Java expression*
a b–c	a*b–c
(m+n)(x+y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c
$3x^2+2x+1$	3*x*x+2*x+1
$\frac{x}{y}+c$	x/y+c

Evaluation of Expression

Evaluation of Expressions Expressions are evaluated using an assignment statement of the form

variable = expressions;

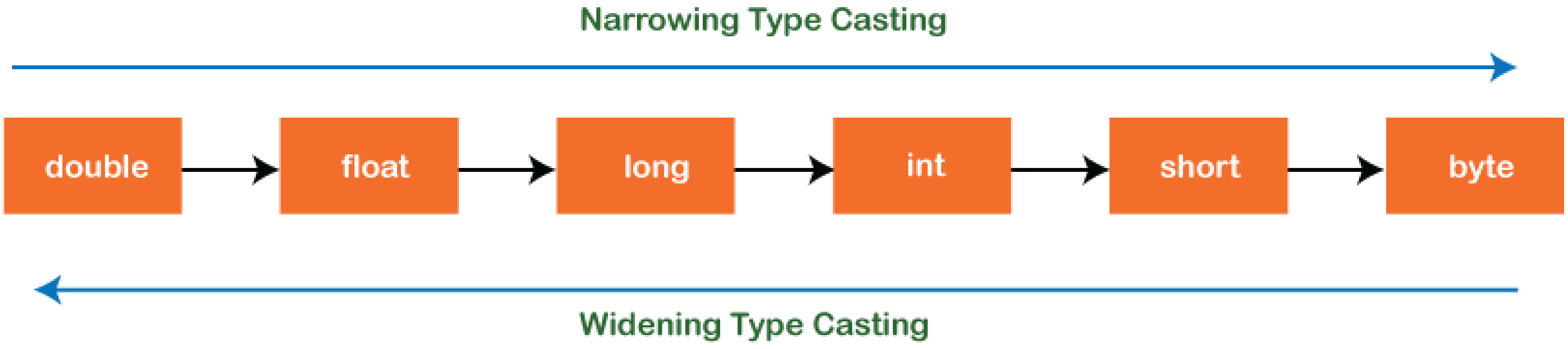
variable is any valid Java variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned value before evaluation is attempted. Examples of evaluation statements are

x = a*b–c;
y = b/c*a;
z = a–b/c+d;

The blank space around an operator is optional and is added only to improve readability. When these statements are used in program, the variables a,b,c and d must be defined before they are used in the expressions.

Type Casting

Type casting in Java is a fundamental concept that allows developers to convert data from one data type to another. It is essential for handling data in various situations, especially when dealing with different types of variables, expressions, and methods. In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.



Type Casting in Java

Convert a value from one data type to another data type is known as **type casting**.

Rules of Typecasting

Widening Conversion (Implicit)

No explicit notation is required.

Conversion from a smaller data type to a larger data type is allowed.

No risk of data loss.

Narrowing Conversion (Explicit)

Requires explicit notation using parentheses and casting.

Conversion from a larger data type to a smaller data type is allowed.

Risk of data loss due to truncation.

Use Cases

Typecasting is commonly used in various scenarios, such as:

Converting between primitive data types.

Handling data in expressions and calculations.

Interacting with different methods and APIs that expect specific data types.

Types of Type Casting

There are two types of type casting:

Widening Type Casting

Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

Both data types must be compatible with each other.

The target type must be larger than the source type.

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

Why Widening Type Casting?

Widening conversion needs to be implemented in order to enable Java to work smoothly with different data types. It creates unbroken workflows when an element of a smaller type is used in a context that needs a larger type. The reason for generalizing the narrower type is in order not to lose any data by converting the smaller type to the larger one, and preserving the whole information.

Types of Widening Type Casting

The common procedure of the Widening type casting is about conversion from primitive to primitive data types in Java.

From byte to short, int, long, float, or double.

From data to type int, long, float, or double.

Char to int, long, float, or double can be converted.

Various other types like int, long, float, or double can also be used.

Key Points to Note

Widening typecasting is performed automatically by the Java compiler when converting from a smaller data type to a larger data type.

No explicit notation, such as casting, is required for widening typecasting.

Widening conversions are always safe and do not result in any loss of data.

Widening typecasting is commonly used in assignments, expressions, and method invocations where data of different types interact.

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other.

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting, then the compiler reports a compile-time error.

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

Why Narrowing Type casting?

Narrowing typecasting becomes necessary when we need to convert data from a larger data type to a smaller one. It often occurs when we are working with data of different sizes and need to fit larger values into smaller containers.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

Command Line Argument

The Java command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple Example of Command Line Arguments in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

Example

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}
}
```

```
compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo
```

Output:

```
Your first argument is: sonoo
```

Scanner Class

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

Example

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

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

    // Enter username and press Enter
    System.out.println("Enter username");
    userName = myObj.nextLine();

    System.out.println("Username is: " + userName);
  }
}
```

Java Scanner Methods to Take Input

The Scanner class provides various methods that allow us to read inputs of different types.

Method	Description
nextInt()	reads an int value from the user
nextFloat()	reads a float value form the user
nextBoolean()	reads a boolean value from the user
nextLine()	reads a line of text from the user
next()	reads a word from the user
nextByte()	reads a byte value from the user
nextDouble()	reads a double value from the user
nextShort()	reads a short value from the user
nextLong()	reads a long value from the user

Example of different methods to read data

```
import java.util.Scanner;

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

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

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

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

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

Conditional statement

The Java if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in Java.

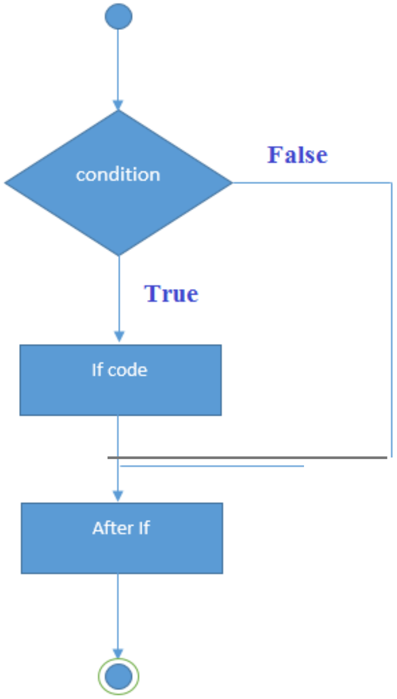
if statement
if-else statement
if-else-if ladder
nested if statement

if Statement

The Java if statement tests the condition. It executes the if block if condition is true.

Syntax:

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



Example

```
//Java Program to demonstate the use of if statement.
public class Main {
public static void main(String[] args) {
    //defining an 'age' variable
    int age=20;
    //checking the age
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}
```

Output

```
Age is greater than 18
```

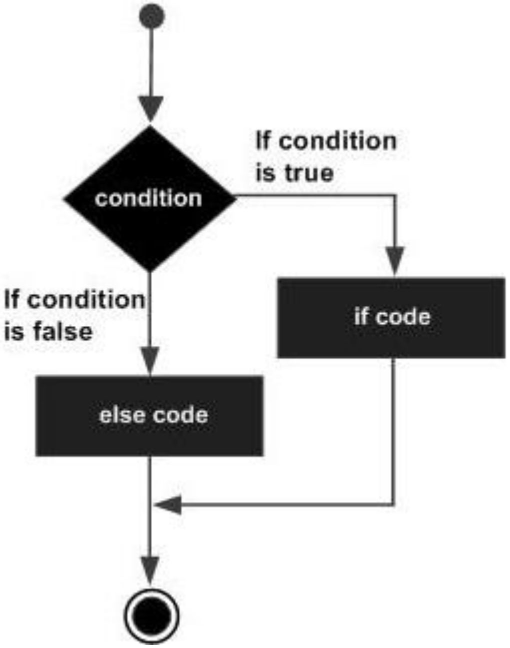
if-else Statement

The if-else statement allows Java programs to handle both true and false conditions. If the condition inside the if statement evaluates to false, the else block is executed instead.

Using if-else statements in Java improves decision-making in programs by executing different code paths based on conditions.

Syntax of if-else Statement

```
if(Boolean_expression)
{
// Executes when the Boolean expression is true
}else {
// Executes when the Boolean expression is false
}
```



Example

```
//Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class Main {
public static void main(String[] args) {
    //defining a variable
    int number=13;
    //Check if the number is divisible by 2 or not
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}
```

Output

```
Odd number
```

Ladder if (if else if) Statements

The **if...else if...else statement** is used for executing multiple code blocks based on the given conditions (Boolean expressions).

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using a single **if...else if** statement.

Syntax

if(Boolean_expression 1) { // Executes when the Boolean expression 1 is true }else
if(Boolean_expression 2) { // Executes when the Boolean expression 2 is true }else
if(Boolean_expression 3) { // Executes when the Boolean expression 3 is true }else { // Executes when the none of the above condition is true. }

Fig: else-if ladder

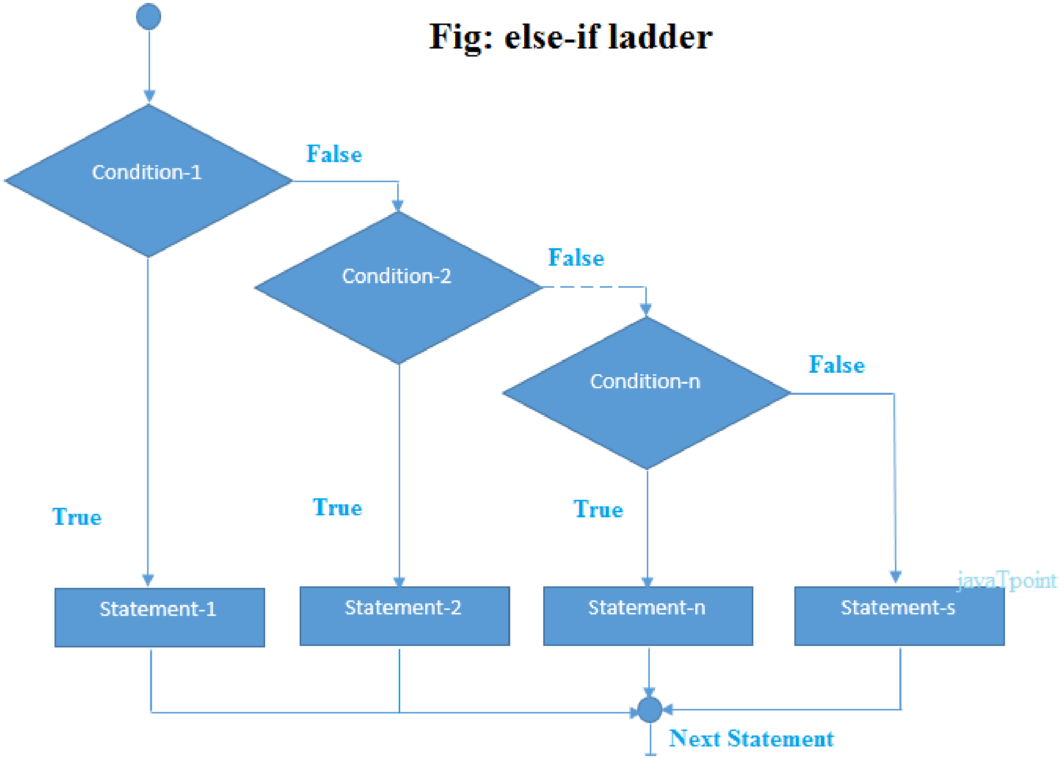
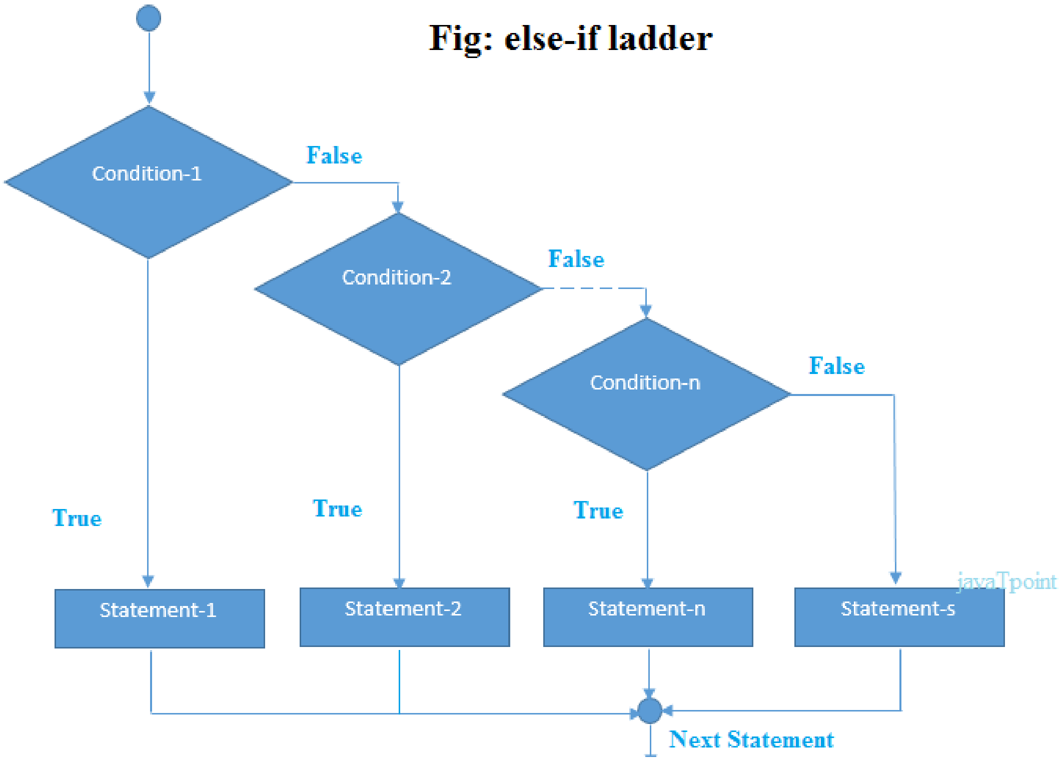


Fig: else-if ladder



Example

```
public class Main {
public static void main(String[] args) {
    int marks=65;

    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

Program to check POSITIVE, NEGATIVE or ZERO using if-else-if:

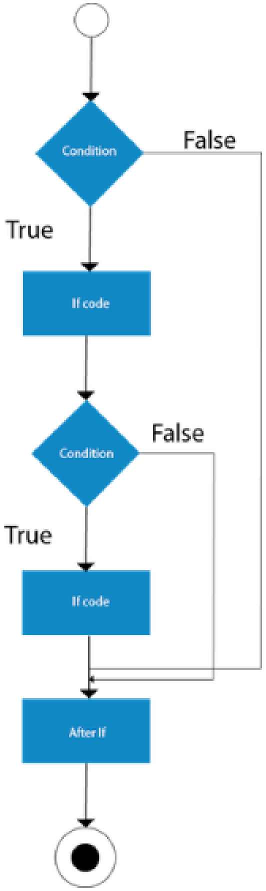
```
public class Main {
public static void main(String[] args) {
    int number=-13;
    if(number>0){
    System.out.println("POSITIVE");
    }else if(number<0){
    System.out.println("NEGATIVE");
    }else{
    System.out.println("ZERO");
   }
}
}
```

Nested if-else Statement

The nested if else statement is used for better decision-making when other conditions are to be checked when a given condition is true. In the nested if else statement, you can have an if-else statement block the another if (or, else) block.

Syntax

```
if(condition1){
// code block
if(condition2){
//code block
}
}
```



Example 1

```
public class Main {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=20;
    int weight=80;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        }
    }
}}
```

Example 2

```
//Java Program to demonstrate the use of Nested If Statement.
public class Main {
public static void main(String[] args) {
    //Creating two variables for age and weight
    int age=25;
    int weight=48;
    //applying condition on age and weight
    if(age>=18){
        if(weight>50){
            System.out.println("You are eligible to donate blood");
        } else{
            System.out.println("You are not eligible to donate blood");
        }
    } else{
      System.out.println("Age must be greater than 18");
    }
}
}
```

Switch statement

The *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.

The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, we can use strings in the switch statement.

The switch statement can be described as control flow type statement which is utilized for manipulating the flow of program execution and invoking various branches of code using the value of an expression.

In other words, the switch statement tests the equality of a variable against multiple values.

Points to Remember

There can be *one or N number of case values* for a switch expression.

The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.

The case values must be *unique*. In case of duplicate value, it renders compile-time error.

The Java switch expression must be of *byte*, *short*, *int*, *long (with its Wrapper type)*, *enums and string*.

Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.

The case value can have a *default label* which is optional.

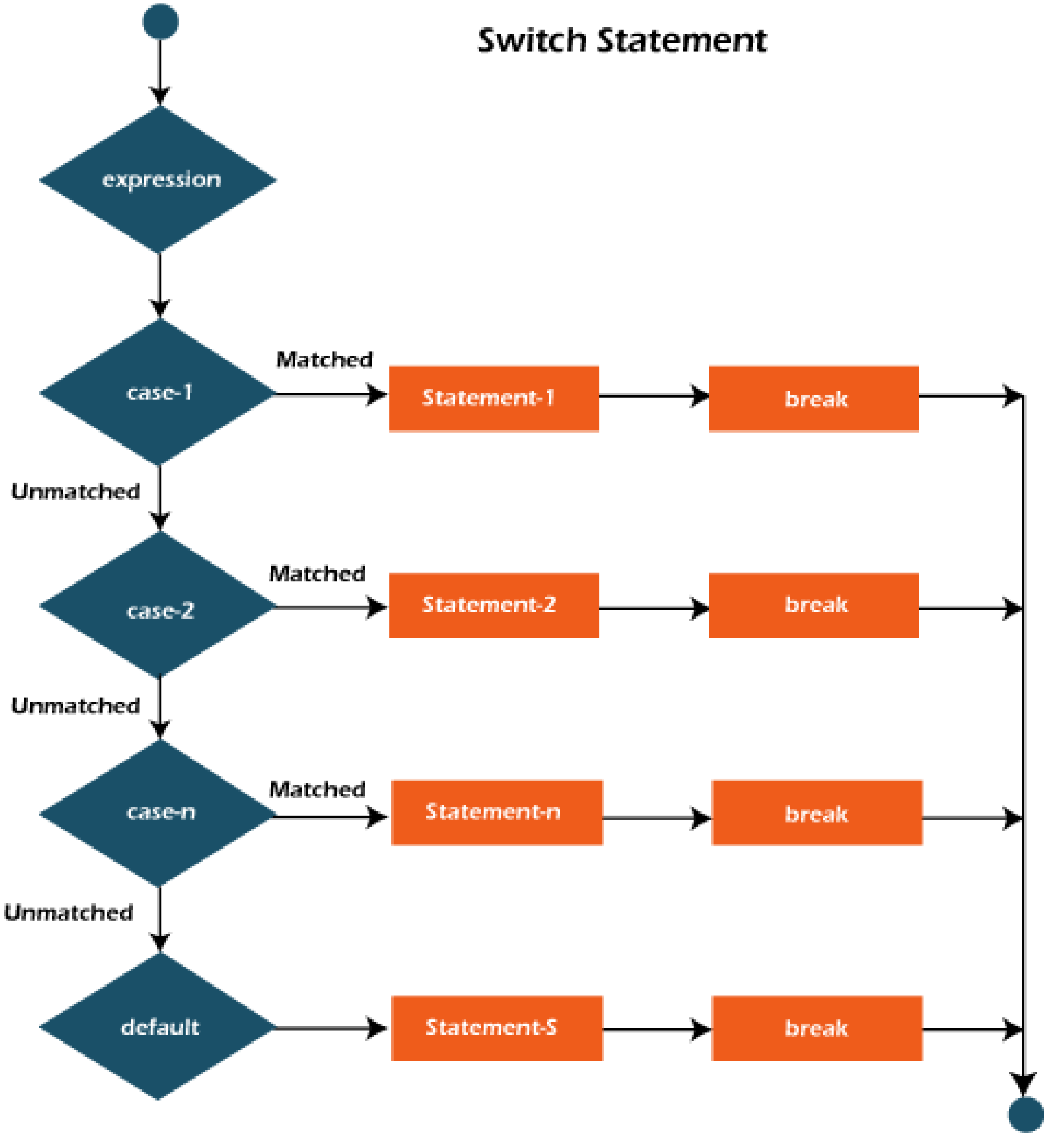
In Java, switch statement mainly provides a more detailed alternative that avoids the usage of nested or several if-else statements when associated with an individual variable.

The syntax of the Java switch statement contains the **switch** keyword which is followed by the expression that needs to be evaluated using parentheses. The mentioned expression must definitely evaluate to a definite data type which is primitive such as int, char, or enum.

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
  code to be executed if all cases are not matched;
}
```



In Java, the switch statement can also contain a default label. The default label will be executed only in the situation when none of the case labels are matching the expressions value. The declaring of default label is considered optional, but can be useful in the events of unexpected values or inputs.

```
public class Main {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Looping statement

In programming, loops play a pivotal role in iterating over a set of statements repeatedly until a specific condition is met. One such loop in Java is the 'while' loop, known for its simplicity and versatility.

Types of Loops

While Loop

Do while Loop

For Loop

The Java while loop is used to iterate a part of the program repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while loop.

Syntax:

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

Here, condition is a boolean expression that determines whether the loop should continue iterating or not. The statements within the curly braces are executed repeatedly as long as the condition evaluates to true.

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. When the condition becomes false, we exit the while loop.

Example:

```
i <=100
```

2. Update Expression: Every time the loop body is executed, this expression increments or decrements loop variable.

Example:

```
i++;
```

Basic Usage

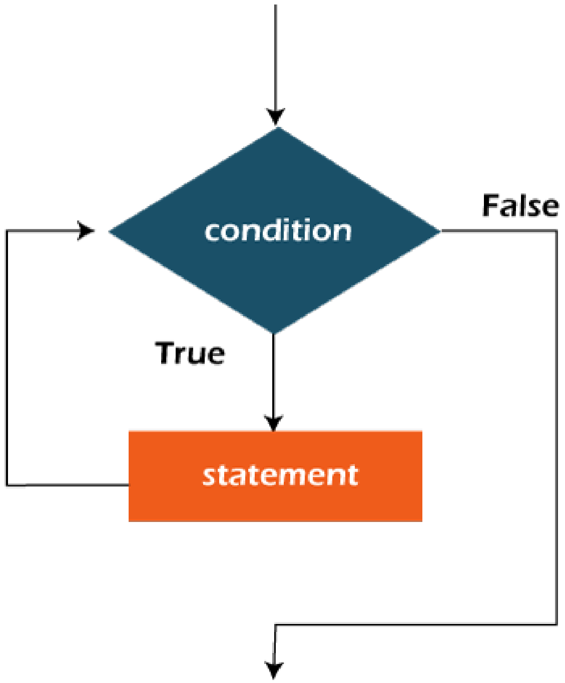
Let's delve into a simple example to grasp the fundamental usage of a while loop. Consider a scenario where we want to print numbers from 1 to 5:

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

In this example, the loop starts with i initialized to 1. The condition i <= 5 ensures that the loop continues as long as i is less than or equal to 5. Within each iteration, i is incremented by 1, ensuring that the loop doesn't become infinite.

Java While Loop Flowchart

Here, the important thing about while loop is that, sometimes it may not even execute. If the condition to be tested results into false, the loop body is skipped and first statement after the while loop will be executed.



Example

```
public class Main {
public static void main(String[] args) {
    int i=1;
    while(i<=10){
        System.out.println(i);
    i++;
    }
}
}
```

```
// Java Program to print factorial of 5 using while loop
public class Main {
    public static void main(String[] args) {
        // Declare a variable to 5. This is the number whose factorial is to be calculated.
        int number = 5;
        // Declare a variable 'factorial' and initialize it to 1. This variable will hold the result of the factorial c
alculation.
        int factorial = 1;
        // Declare a variable 'i' and initialize it to 1.
        int i = 1;
        //Start a while loop
        while( i <= number ) {
            // Multiply the current value of 'factorial' by 'i' and store the result back in 'factorial'.
            factorial *= i; // This is equivalent to factorial = factorial * i;
            i++;
        }
        // Print the calculated factorial to the console.
        System.out.println("Factorial of " + number + " is: " + factorial);
    }
}
```

do-while Loop

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed / loop body
//update statement
}while (condition);
```

The different parts of do-while loop:

1. Condition: It is an expression which is tested. If the condition is true, the loop body is executed and control goes to update expression. As soon as the condition becomes false, loop breaks automatically.

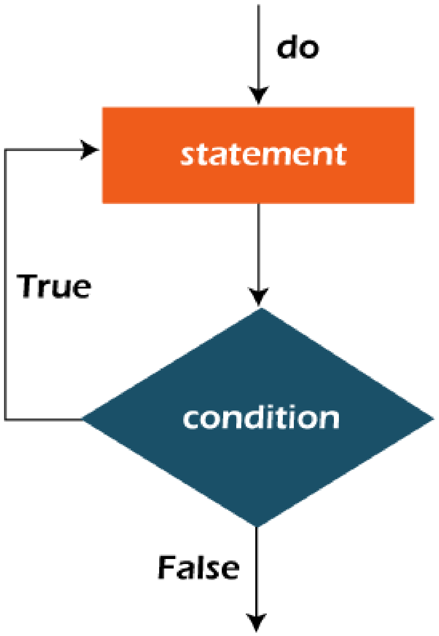
i <=100

2. Update expression: Every time the loop body is executed, the this expression increments or decrements loop variable.

i++;

Note: The do block is executed at least once, even if the condition is false.

Flow chart



```
//Simple do-while example in Java
public class Main {
public static void main(String[] args) {
    //initialization
    int i=1;
    //do-while loop
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

For loop

For loops in Java are a fundamental control structure used to repeat a block of code a specific number of times or iterate through a sequence of values. They are incredibly useful for tasks that require repetition, such as processing items in an array, generating repetitive output, or executing a block of code a predetermined number of times.

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is **fixed**, it is recommended to use for loop.

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

Initialization: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.

Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.

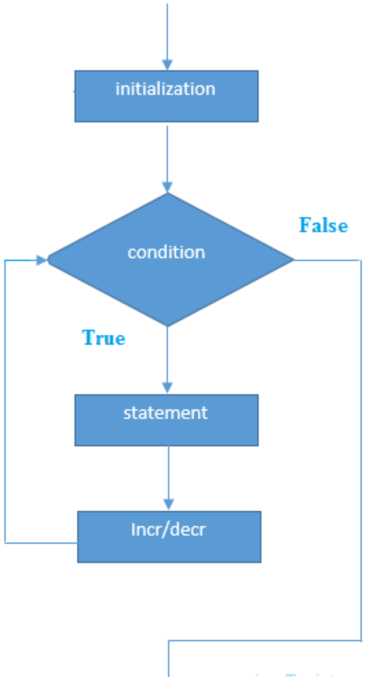
Increment/Decrement: It increments or decrements the variable value. It is an optional condition.

Statement: The statement of the loop is executed each time until the second condition is false.

Syntax:

```
for(initialization; condition; increment/decrement){
//statement or code to be executed
}
```

Flowchart



```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class Main {
public static void main(String[] args) {
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```

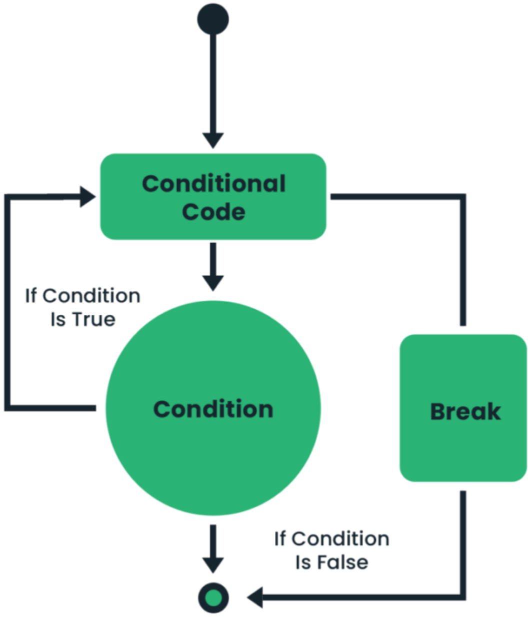
Jump statement

Jump statements in Java provide a way to modify the normal flow of control within loops or switch statements. They allow you to jump to a specific point in your code, skip iterations of a loop, or exit a loop or method altogether.

Break Statement

The ‘break’ statement in Java is a jump statements that allows you to exit a loop or switch statement prematurely. It provides a way to break out of the current code block and continue execution outside of it.

Flow Chart of Break Statement



Example

```
public class BreakExample {
    public static void main(String[] args) {
        int choice = 2;

        switch (choice) {
            case 1:
                System.out.println("You selected option 1.");
                break;
            case 2:
                System.out.println("You selected option 2.");
                break;
            case 3:
                System.out.println("You selected option 3.");
                break;
            default:
                System.out.println("Invalid choice.");
                break;
        }
        System.out.println("End of program.");
    }
}
```

Output

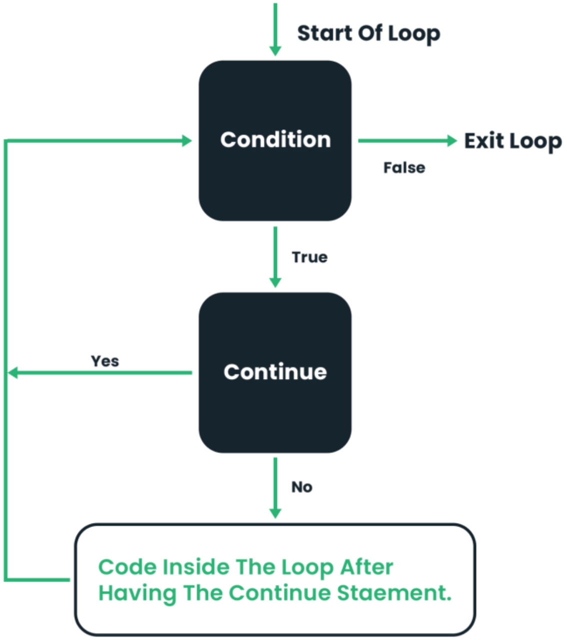
```
You selected option 2.
```

Continue Statement

In Java, the continue statement is used to skip the remaining code in a loop (will study later in the next blog) iteration and go to the next iteration. It allows you to bypass specific parts of the loop's code block based on a condition.

When a given condition is met, the continue statement in Java programming allows you to skip the remaining code within a loop's iteration. It allows you to manage scenarios when you want to bypass specific iterations and continue with the next iteration of the loop more efficiently. You can regulate the flow of execution and optimize your code based on certain conditions or requirements in this manner.

Flowchart of the Continue Statement



```
public class ContinueExample {
    public static void main(String[] args) {
        // Imagine you are counting from 1 to 10
        for (int i = 1; i <= 10; i++) {
            // Check if the current number is divisible by 2
            if (i % 2 == 0) {
                // Skip the iteration if the number is divisible by 2
                continue;
            }

            // Print the current number
            System.out.println(i);
        }
    }
}
```

Output

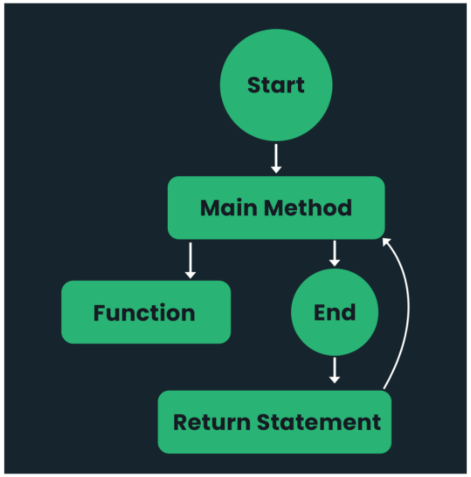
```
1
3
5
7
9
```

Return Statement

In Java, the return statement is used to exit a method and provide a result or value back to the caller. It’s like completing a task and giving something in return.

For example, let’s say you have a method called calculateTotalPrice that calculates the total price of a pizza order. After performing the necessary calculations, you use the return statement to send the final price back to the code that called the method. This way, the calling code can use the returned value for further processing or display.

Flow Chart



Example

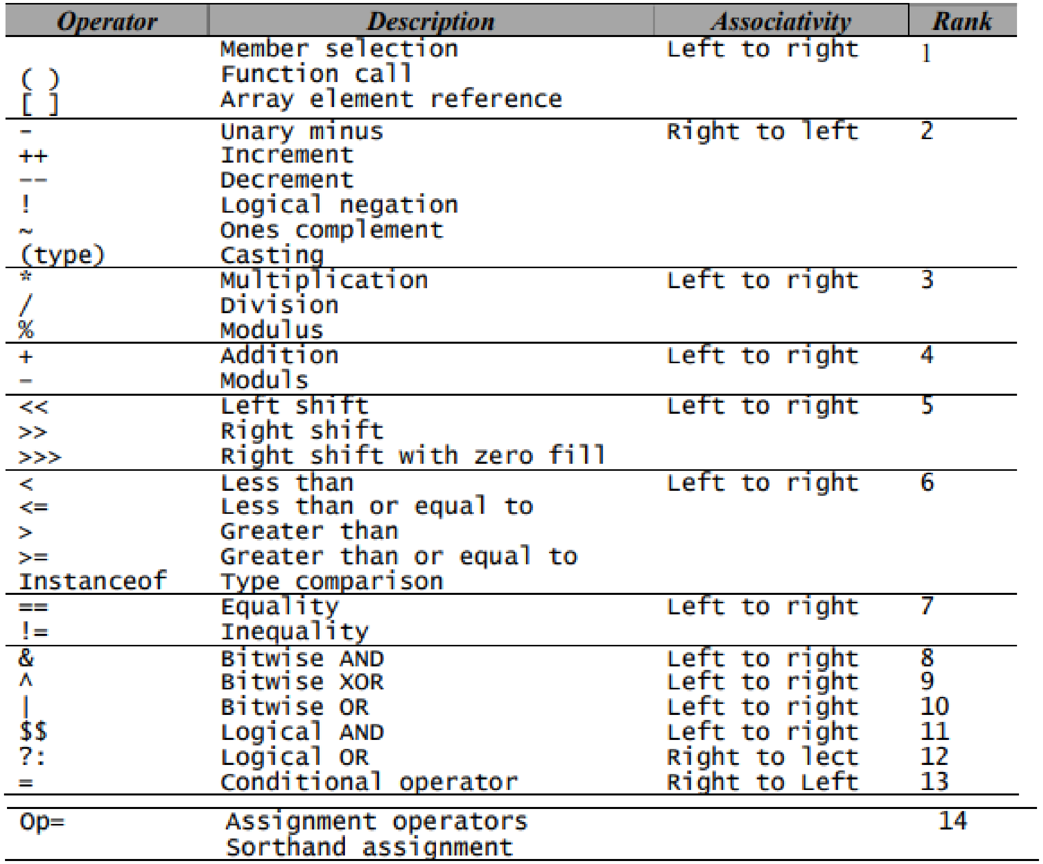
```
public class PizzaDelivery {
    public static double calculateTotalPrice(int pizzaCount) {
        double pricePerPizza = 12.99;
        double totalPrice = pizzaCount * pricePerPizza;
        // Return the calculated total price
        return totalPrice;
    }
    public static void main(String[] args) {
        int numberOfPizzas = 3;
        double total = calculateTotalPrice(numberOfPizzas);
        System.out.println("Total price: $" + total);
    }
}
```

Output

```
Total price: 38.97
```

Operator Precedence & Associativity

Each operator in Java has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of the levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as the associativity property of an operator. Table 3.11 provides a complete lists of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence (rank 1 indicates the highest precedence level and 14 the lowest).



Operator	Description	Associativity	Rank
	Member selection	Left to right	1
( )	Function call		
[ ]	Array element reference		
-	Unary minus	Right to left	2
++	Increment		
--	Decrement		
!	Logical negation		
~	Ones complement		
(type)	Casting		
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
-	Moduls		
<<	Left shift	Left to right	5
>>	Right shift		
>>>	Right shift with zero fill		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
Instanceof	Type comparison		
==	Equality	Left to right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
$$	Logical AND	Left to right	11
?:	Logical OR	Right to lect	12
=	Conditional operator	Right to Left	13
Op=	Assignment operators Sorthand assignment		14

It is very important to note carefully, the order of precedence and associativity of operators. Consider the following conditional statement:

if(x == 10+15 && y<10)

The precedence rules say that the addition operator has a higher priority than the logical operator (&&) and the relational operator (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

if(x == 25 &&y<10)

The next step is to determine whether x is equal to 25 and y is less than 10. If we assume a value fo 20 for x and 5 for y, then

x == 25 is FALSE

y < 10 is TRUE

Note that since the operator < enjoys a higher priority compared to ==, y<10 is tested first and then x = = 25 is tested.

Finally we get:

if(FALSE && TRUE)

Because one of the conditions FALSE, the compound condition if FALSE.