Semester III
BCA311 JAVA Programming (Unit-2)

Harshita Mathur
Department of Computer Science
Email : hmathur85@gmail.com

Command Line Arguments

- A command-line argument is an information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to main( ).

Example

```
public class my
 {    public static void main(String args[])
    {
     for(int i = 0; i<args.length; i++)
      {
         System.out.println(args[i]);
       }
     }
}
```

Comliping and executing..

- **Javac** *my.java*
- **Java** *my* this is a command line

Output:

```
this
 is
a
 command
line
```

Class Fundamentals

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity.
- A class in Java can contain:
- **Fields**
- **Methods**
- **Constructors**
- **Nested class and interface**

Syntax to declare a class:

```
class <class_name>
{
field;
method;
}
```

Instance variable in Java

- A variable which is created inside the class but outside the method is known as an instance variable.

- Instance variable doesn't get memory at compile time.

- It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

- In Java, a method is like a function which is used to expose the behavior of an object.

- Advantage of Method:

  - Code Reusability
  - Code Optimization

Object and Class Example: main within the class

```
•  //Java Program to illustrate how to define a class and fields
•  //Defining a Student class.
•  class Student{
•   //defining fields
•   int id;//field or data member or instance variable
•   String name;
•   //creating main method inside the Student class
•   public static void main(String args[]){
•    //Creating an object or instance
•    Student s1=new Student();//creating an object of Student
•    //Printing values of the object
•    System.out.println(s1.id);//accessing member through reference variable
•    System.out.println(s1.name);
•   }
•  }
```

Output:

```
0
null
```

Object and Class Example: main outside the class

- //Java Program to demonstrate having the main method in
- //another class
- //Creating Student class.
- **class** Student{
- **int** id;
- String name;
- }
- //Creating another class TestStudent1 which contains the main method
- **class** TestStudent1{
- **public static void** main(String args[]){
-   Student s1=**new** Student();
-   System.out.println(s1.id);
-   System.out.println(s1.name);
- }
- }

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

- By reference variable
- By method
- By constructor

Object and Class Example: Initialization through reference

- **class** Student{
- **int** id;
- String name;
- }
- **class** TestStudent2{
- **public static void** main(String args[]){
-   Student s1=**new** Student();
-   s1.id=101;
-   s1.name=“Ravi";
-   System.out.println(s1.id+" "+s1.name);//printing members with a white space
- }
- }

- Output:

```
101 Ravi
```

We can also create multiple objects and store information in it through reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name=“Ravi";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

- Output:

```
101 Ravi
102 Amit
```

Object and Class Example: Initialization through method

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

- Output:

```
111 Karan
222 Aryan
```

Object and Class Example: Initialization through a constructor

- **class** Employee{
-    **int** id;
-    String name;
-    **float** salary;
-    **void** insert(**int** i, String n, **float** s) {
-       id=i;
-       name=n;
-       salary=s;
-    }
-    **void** display(){System.out.println(id+" "+name+" "+salary);}
- }
- **public class** TestEmployee {
- **public static void** main(String[] args) {
-    Employee e1=**new** Employee();
-    Employee e2=**new** Employee();
-    Employee e3=**new** Employee();
-    e1.insert(101,"ajeet",45000);
-    e2.insert(102,"irfan",25000);
-    e3.insert(103,"nakul",55000);
-    e1.display();
-    e2.display();
-    e3.display();
- }
- }

Output:

```
 101 ajeet 45000.0
 102 irfan 25000.0
 103 nakul 55000.0
```

Tokens

- In Java, a program is a collection of classes and methods, while methods are a collection of various expressions and statements.
- Tokens in Java are the small units of code which a Java compiler uses for constructing those statements and expressions. Java supports 5 types of tokens which are:
- Keywords
- Identifiers
- Literals
- Operators
- Special Symbols

Keywords

- Keywords in Java are predefined or reserved words that have special meaning to the Java compiler.
- Each keyword is assigned a special task or function and cannot be changed by the user.
- You cannot use keywords as variables or identifiers as they are a part of Java syntax itself.
- A keyword should always be written in lowercase as Java is a case sensitive language. Java supports various keywords, some of them are listed below:

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. throw	40. throws
41. transient	42. try	43. void	44. volatile	45. while
46. assert	47. const	48. enum	49. goto	50. strictfp

Identifier

- Java Identifiers are the user-defined names of variables, methods, classes, arrays, packages, and interfaces. Once you assign an identifier in the Java program, you can use it to refer the value associated with that identifier in later statements. There are some standards which you must follow while naming the identifiers such as:
- Identifiers must begin with a letter, dollar sign or underscore.
- Apart from the first character, an identifier can have any combination of characters.
- Identifiers in Java are case sensitive.
- Java Identifiers can be of any length.
- Identifier name cannot contain white spaces.
- Any identifier name must not begin with a digit but can contain digits within.
- Most importantly, **keywords** can’t be used as identifiers in Java.

Literals

- Literals in Java are similar to normal variables but their values cannot be changed once assigned. In other words, literals are constant variables with fixed values. These are defined by users and can belong to any data type. Java supports five types of literals which are as follows:
- Integer
- Floating Point
- Character
- String
- Boolean

Operator

- An operator in Java is a special symbol that signifies the compiler to perform some specific mathematical or non-mathematical operations on one or more operands. Java supports 8 types of operators.

Operator	Examples
Arithmetic	+ , - , / , * , %
Unary	++ , - - , !
Assignment	= , += , -= , *= , /= , %= , ^=
Relational	==, != , < , >, <= , >=
Logical	&& , ||
Ternary	(Condition) ? (Statement1) : (Statement2);
Bitwise	& , | , ^ , ~
Shift	<< , >> , >>>

Special Symbols

- Special symbols in Java are a few characters which have special meaning known to Java compiler and cannot be used for any other purpose.

Symbol	Description
Brackets []	These are used as an array element reference and also indicates single and multidimensional subscripts
Parentheses()	These indicate a function call along with function parameters
Braces{}	The opening and ending curly braces indicate the beginning and end of a block of code having more than one statement
Comma (,)	This helps in separating more than one statement in an expression
Semi-Colon (;)	This is used to invoke an initialization list
Asterisk (*)	This is used to create a pointer variable in Java

Primitive Data Types

- **8 Primitive Data Types**
- **boolean**
- The boolean data type has two possible values, either true or false.
- Default value: false.
- They are usually used for true/false conditions. For example,

```
class BooleanExample
{
  public static void main(String[] args)
  {
    boolean flag = true;
      System.out.println(flag);
   }
 }
```

Output:

```
true
```

- **byte**
- The byte data type can have values from -128 to 127 (8-bit signed two's complement integer).
- It's used instead of int or other integer data types to save memory if it's certain that the value of a variable will be within [-128, 127].
- Default value: 0
- Example:

```
  class ByteExample
{
           public static void main(String[] args)
          {
             byte range;
             range = 124;
             System.out.println(range);
          }
}
```

Output:

```
124
```

- **short**
- The short data type can have values from -32768 to 32767 (16-bit signed two's complement integer).
- It's used instead of other integer data types to save memory if it's certain that the value of the variable will be within [-32768, 32767].
- Default value: 0
- Example:

```
class ShortExample
{
  public static void main(String[] args)
  {
            short temperature;
            temperature = -200;
            System.out.println(temperature);
  }
}
```

When you run the program, the output will be:

```
-200
```

- **int**
- The int data type can have values from -2^{31} to $2^{31}-1$ (32-bit signed two's complement integer).
- If you are using Java 8 or later, you can use unsigned 32-bit integer with a minimum value of 0 and a maximum value of $2^{32}-1$.
- Default value: 0
- Example:

```
class IntExample
 {
        public static void main(String[] args)
       {
        int range = -4250000;
       System.out.println(range);
        }
 }
```

Output:

```
-4250000
```

- **long**
- The long data type can have values from -2^{63} to 2^{63}-1 (64-bit signed two's complement integer).
- If you are using Java 8 or later, you can use unsigned 64-bit integer with a minimum value of 0 and a maximum value of 2^{64}-1.
- Default value: 0
- Example:
- class LongExample

```
class LongExample
{
 public static void main(String[] args)
{
long range = -42332200000L;
System.out.println(range);
 }
 }
```

Output:

-42332200000 Notice, the use of L at the end of -42332200000. This represents that it's an integral literal of the long type.

- **double**
- The double data type is a double-precision 64-bit floating-point.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0d)
- Example:

```
class DoubleExample
 {
 public static void main(String[] args)
 {
         double number = -42.3;
         System.out.println(number);
 }
 }
```

Output:

```
-42.3
```

- **float**
- The float data type is a single-precision 32-bit floating-point.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0f)
- Example:

```
class FloatExample
 {
  public static void main(String[] args)
  {
  float number = -42.3f;
 System.out.println(number);
 }
  }
```

Output:

```
-42.3
```

- Notice that, we have used -42.3f instead of -42.3in the above program. It's because -42.3 is a double literal. To tell the compiler to treat -42.3 as float rather than double, you need to use f or F.

- **char**
- It's a 16-bit Unicode character.
- The minimum value of the char data type is '\u0000' (0). The maximum value of the char data type is '\uffff'.
- Default value: '\u0000'
- Example:
- class CharExample

```
 {
            public static void main(String[] args)
             {
             char letter = '\u0051';
            System.out.println(letter);
             }
 }
```

Output:

```
Q
```

- You get the output Q because the Unicode value of Q is '\u0051'.

- Here is another example:

```
class CharExample
 {
 public static void main(String[] args)
 {
           char letter1 = '9';
          System.out.println(letter1);
          char letter2 = 65;
          System.out.println(letter2);
 }
 }
```

Output:

```
9
A
```

- When you print letter1, you will get 9 because letter1 is assigned character '9'. When you print letter2, you get A because the ASCII value of 'A' is 65. It's because java compiler treats the character as an integral type.

- **String**
- Java also provides support for character strings via java.lang.String class. Here's how you can create a String object in Java:
- myString = "Programming is awesome";

Operator precedence

- Operator precedence determines the order in which the operators in an expression are evaluated.
- Now, take a look at the statement below:
- int myInt = 12 - 4 * 2;What will be the value of myInt? Will it be (12 - 4)*2, that is, 16? Or it will be 12 - (4 * 2), that is, 4?
- When two operators share a common operand, 4 in this case, the operator with the highest precedence is operated first.
- In Java, the precedence of * is higher than that of -. Hence, the multiplication is performed before subtraction, and the value of myInt will be 4.

Java Operator Precedence

Operators	Precedence
postfix increment and decrement	++ --
prefix increment and decrement, and unary	++ -- + - ~ !
Multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof

Java Operator Precedence

equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Example: Operator Precedence

```
class Precedence
{
 public static void main(String[] args)
 {
   int a = 10, b = 5, c = 1, result;
   result = a-++c-++b;
  System.out.println(result);
  }
}
```

Output:

```
2
```

Associativity of Operators in Java

- If an expression has two operators with similar precedence, the expression is evaluated according to its associativity (either left to right, or right to left). Let's take an example.

  a = b = c

- Here, the value of c is assigned to variable b. Then the value of b is assigned of variable a. Why? It's because the associativity of = operator is from right to left.

Java Operator Precedence and Associativity

Operators	Precedence	Associativity
postfix increment and decrement	++ --	left to right
prefix increment and decrement, and unary	++ -- + - ~ !	right to left
multiplicative	* / %	left to right
additive	+ -	left to right

Java Operator Precedence and Associativity

shift	<< >> >>>	left to right
relational	< > <= >= instanceof	left to right
equality	== !=	left to right
bitwise AND	&	left to right
bitwise exclusive OR	^	left to right
bitwise inclusive OR	|	left to right

Java Operator Precedence and Associativity

logical AND	&&	left to right
logical OR	||	left to right
ternary	? :	right to left
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=	left to right

Semester III
BCA311 JAVA Programming
(Unit-2)

Harshita Mathur

Department of Computer Science

Email : hmathur85@gmail.com

Type Casting/type conversion

- Converting one primitive datatype into another is known as type casting (type conversion) in Java. You can cast the primitive datatypes in two ways namely, Widening and, Narrowing.
- Widening − Converting a lower datatype to a higher datatype is known as widening.
- In this case the casting/conversion is done automatically therefore, it is known as implicit type casting.
- In this case both datatypes should be compatible with each other.

```
class Test
{
    public static void main(String[] args)
    {
        int i = 100;

        // automatic type conversion
        long l = i;

        // automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

- Output:

```
Int value 100
Long value 100
Float value 100.0
```

- **Narrowing** − Converting a higher datatype to a lower datatype is known as narrowing.

- In this case the casting/conversion is not done automatically, you need to convert explicitly using the cast operator “( )” explicitly.

- Therefore, it is known as explicit type casting.

```
•    class Test
•    {
•        public static void main(String[] args)
•        {
•            double d = 100.04;
•
•            //explicit type casting
•            long l = (long)d;
•
•            //explicit type casting
•            int i = (int)l;
•            System.out.println("Double value "+d);
•
•            //fractional part lost
•            System.out.println("Long value "+l);
•
•            //fractional part lost
•            System.out.println("Int value "+i);
•        }
•    }
```

Output:

```
Double value 100.04
Long value 100
Int value 100
```

Accepting Input from Keyboard

There are various ways to accept input from keyboard in java:

- Using Scanner Class
- Using Console Class
- Using InputStreamReader and BufferedReader Class

Accept input using Scanner class in java

```
     import java.util.Scanner; // Needed for Scanner class

    public class RectangleArea
    {
      public static void main(String[] args)
      {
      int length; // To hold rectangle's length.
      int width; // To hold rectangle's width.
      int area; // To hold rectangle's area
     // Create a Scanner object to read input.
     Scanner  sc= new Scanner(System.in);
     // Get length from the user.
     System.out.print("Enter length ");
     length = sc.nextInt();
    // Get width from the user.
      System.out.print("Enter width ");
      width =sc.nextInt();
      // Calculate area.
        area = length * width;
      // Display area.
        System.out.println("The area of rectangle is " + area);
     }
}
```

- **Output :**

```
Enter length 5
Enter width 8
The area of rectangle is 40
```

Some other useful methods of Scanner class

Method	*Returns*
int nextInt()	Returns the next token as an int.
float nextFloat()	Returns the next token as a float.
double nextDouble()	Returns the next token as a long.
String next()	Finds and returns the next complete token as a string ended by a blank.
String nextLine()	Returns the rest of the current line, excluding any line separator at the end.

Example

```
import java.util.Scanner;
        public class ReadEmployee
       {
        public static void main(String[] args)
       {
        String name; // To hold the employee's name
       int age; // To hold the employee's age
       char gender; // To hold the employee's gender
       double salary; // To hold the employee's salary
       Scanner sc= new Scanner(System.in);
        System.out.print("Enter name: ");
         name = sc.nextLine();
        System.out.print("Enter age: ");
         age = sc.nextInt();
        System.out.print("Enter gender: ");
         gender = sc.next().charAt(0);
        System.out.print("Enter salary: ");
        salary = sc.nextDouble();
        System.out.println("Name: " + name + " Age: " + age + " Gender: " + gender + " Salary: " + salary);
      }
      }
```

- Output

```
Enter name: Alex Joseph
Enter age: 24
Enter gender: M
Enter salary: 8000
Name: Alex Joseph Age: 24 Gender: M Salary: 8000.0
```

Accept input using Console class in java

-
  ```
  import java.io.Console;
  class ConsoleClass
  {
  public static void main(String args[])
  {
  Console con = System.console();
  System.out.println("Enter programming skills ");
  /*readLine() is a method of Console class which takes input from
  keyboard */
  String skills = con.readLine();
  System.out.println("Your skills are "+skills);
  }
  }
  ```

Accepting input using InputStreamReader and BufferedReader Class

```
import java.io.*;
class my{
public static void main(String args[])throws Exception{

InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);

System.out.println("Enter your name");
String name=br.readLine();
System.out.println("Welcome "+name);
 }
}
```

Output:

```
 Enter your name
 Amit
Welcome Amit
```

Decision Making

- Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

if statement in java

- An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

```
if(Boolean_expression)
 {
 // Statements will execute if the Boolean expression is true
 }
```

- If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

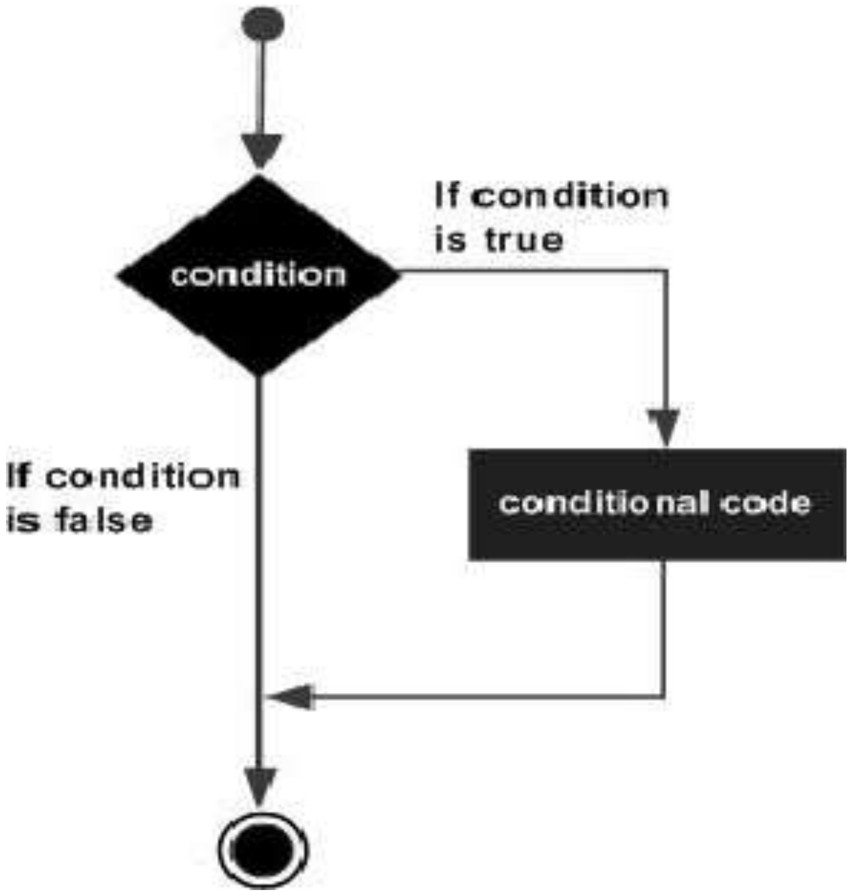
Example

```
public class Test
{
 public static void main(String args[])
 {
 int x = 10;
 if( x < 20 )
 {
 System.out.print("This is if statement");
 }
}
 }
```

Output

```
This is if statement.
```

Flow Diagram



if-else statement in java

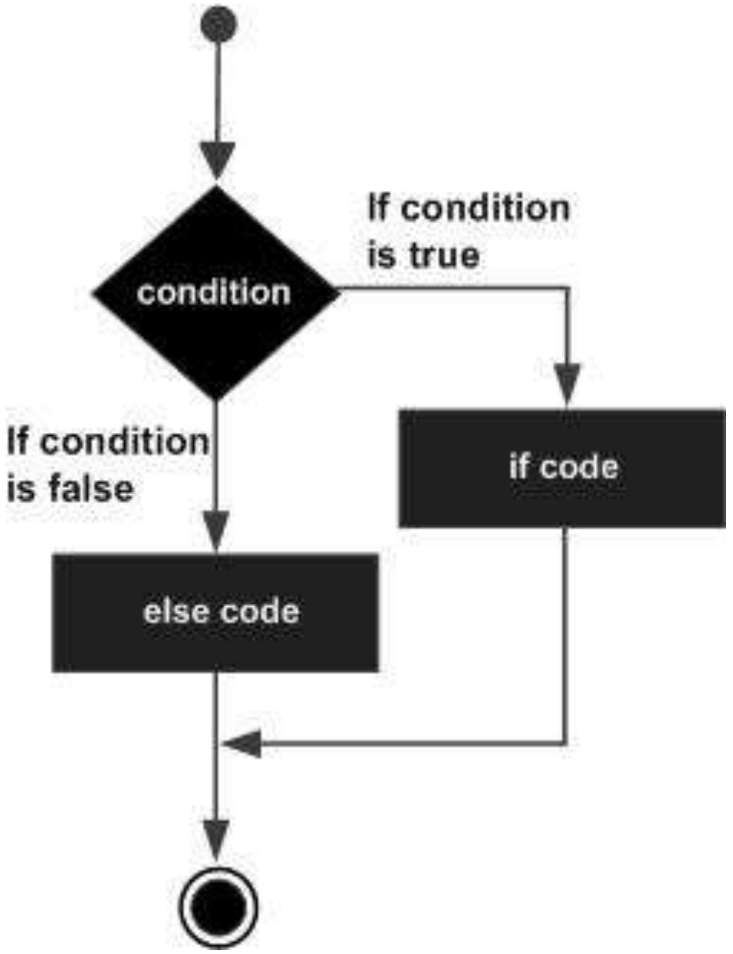
- An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

  Syntax

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

- If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flow Diagram



Example

```
public class Test
 {
 public static void main(String args[])
 {
 int x = 30;
if( x < 20 )
 {
 System.out.print("This is if statement");
}
else
 {
 System.out.print("This is else statement");
}
}
 }
```

Output

```
This is else statement
```

The if…else if…else 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.
}
```

Example

```
public class Test
 {
 public static void main(String args[])
{
 int x = 30;
 if( x == 10 )
 {
 System.out.print("Value of X is 10");
}
else if( x == 20 )
 {
 System.out.print("Value of X is 20");
}
else if( x == 30 )
{
 System.out.print("Value of X is 30");
 }
else
{
System.out.print("This is else statement");
}
 }
 }
```

Output

```
Value of X is 30
```

nested if statement in java

Syntax

```
if(Boolean_expression 1)
{
 // Executes when the Boolean expression 1 is true
if(Boolean_expression 2)
{
 // Executes when the Boolean expression 2 is true
}
 }
```

Example

```
public class Test
{
public static void main(String args[])
 {
 int x = 30;
int y = 10;
if( x == 30 )
{
 if( y == 10 )
 {
System.out.print("X  = 30 and Y = 10");
}
 }
 }
}
```

Output

```
X = 30 and Y = 10
```

switch statement in java

- A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

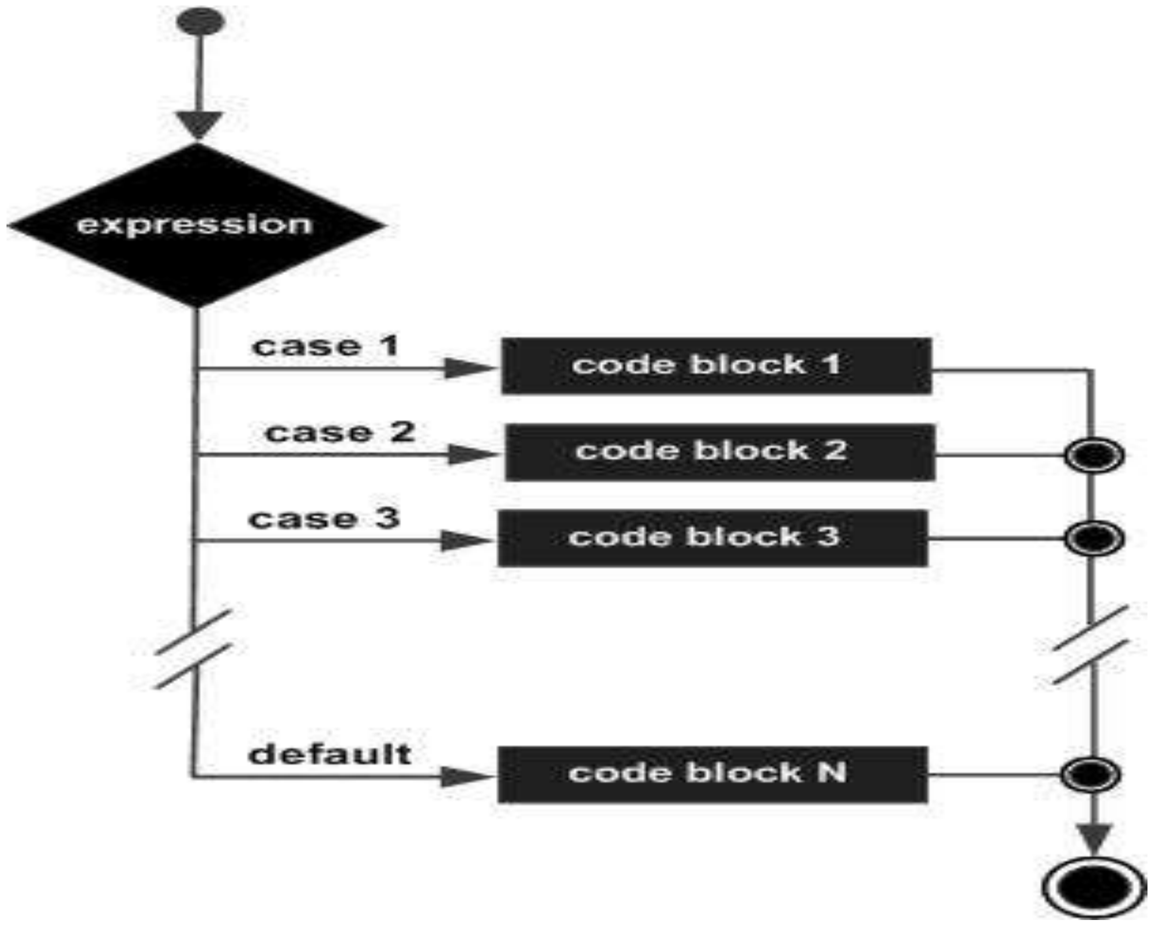
```
switch(expression)
{
case value :
 // Statements
 break; // optional
 case value :
 // Statements
 break; // optional
// You can have any number of case statements.
 default : // Optional
 // Statements
 }
```

The following rules apply to a **switch** statement −

- The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.

- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Example

```
public class Test
{
 public static void main(String args[])
 {
 // char grade = args[0].charAt(0);
 char grade = 'C';
 switch(grade)
{
 case 'A' : System.out.println("Excellent!");
                 break;
 case 'B' : System.out.println("Good");
                  break;
 case 'C' : System.out.println("Well done");
                  break;
case 'D' : System.out.println("You passed");
 case 'F' : System.out.println("Better try again");
                break;
default : System.out.println("Invalid grade");
 }
 System.out.println("Your grade is " + grade); } }
```

Output

```
Well done
Your grade is C
```

Loops in Java

- Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true.

- Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

while loop

- A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.

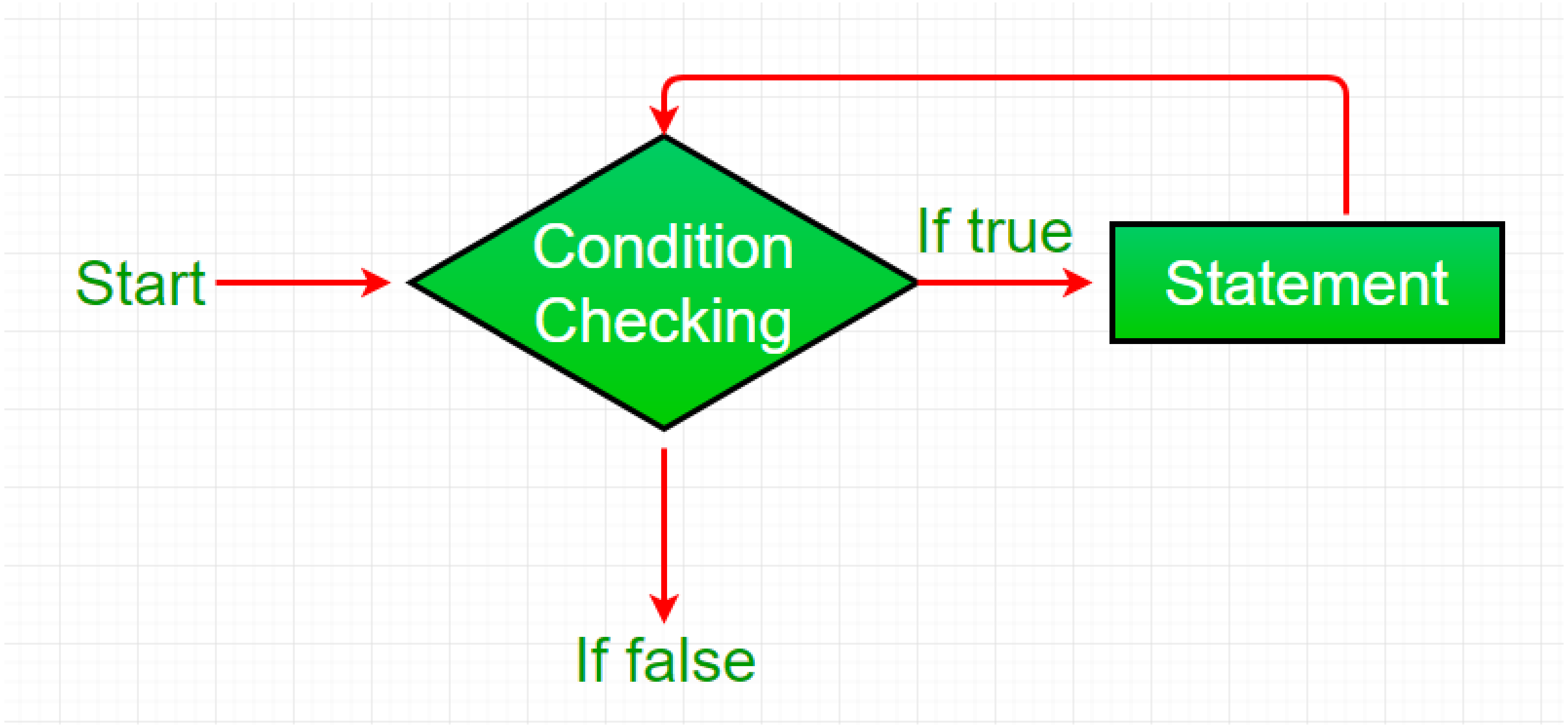
- The while loop can be thought of as a repeating if statement.

  Syntax :

  while (boolean condition)

  { loop statements... }

Flowchart



- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**
- Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

Example

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

        // Exit when x becomes greater than 4
        while (x <= 4)
        {
            System.out.println("Value of x:" + x);

            // Increment the value of x for
            // next iteration
            x++;
        }
    }
}
```

- **Output:**

```
Value of x:1
 Value of x:2
 Value of x:3
 Value of x:4
```

for loop

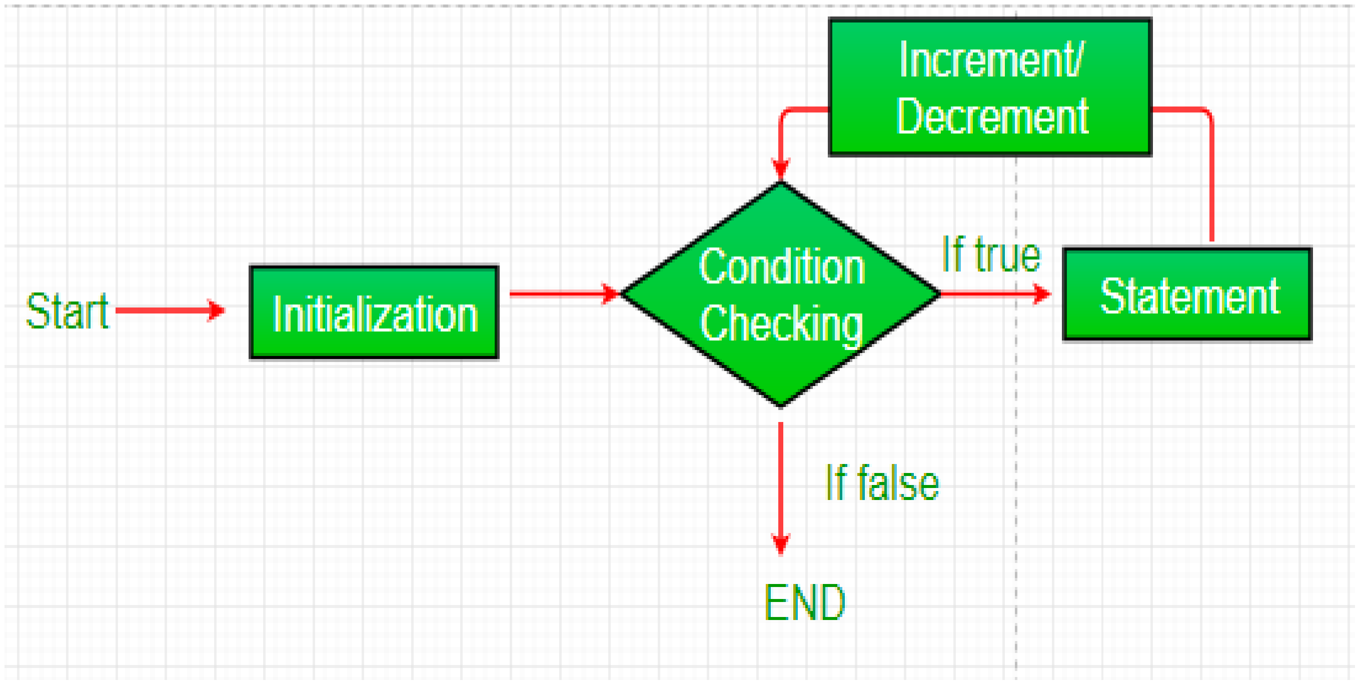
- for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

  Syntax:

  for (initialization condition; testing condition; increment/decrement)

  { statement(s) }

Flowchart


Start
Initialization
Condition Checking
If true
Statement
Increment/ Decrement
If false
END

- **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
- **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
- **Increment/ Decrement:** It is used for updating the variable for next iteration.
- **Loop termination:**When the condition becomes false, the loop terminates marking the end of its life cycle.
-

Example

```
class my
{
    public static void main(String args[])
    {
        // for loop begins when x=2
        // and runs till x <=4
        for (int x = 2; x <= 4; x++)
            System.out.println("Value of x:" + x);
    }
}
```

- **Output:**

```
Value of x:2
Value of x:3
Value of x:4
```

Enhanced For loop

- Java also includes another version of for loop introduced in Java 5.
- Enhanced for loop provides a simpler way to iterate through the elements of a collection or array. It is inflexible and should be used only when there is a need to iterate through the elements in sequential manner without knowing the index of currently processed element.
- It ensures that the values in the array can not be modified, so it can be said as read only loop where you can’t update the values.

Syntax

```
for (T element:Collection obj/array)
        { statement(s) }
```

Example

```
•  public class my
•  {
•      public static void main(String args[])
•      {
•          String array[] = {"Ravi", "Harish", “Hardik"};
•
•          //enhanced for loop
•          for (String x:array)
•          {
•              System.out.println(x);
•          }
•
•      }
•  }
```

- Output:

```
Ravi
Harish
Hardik
```

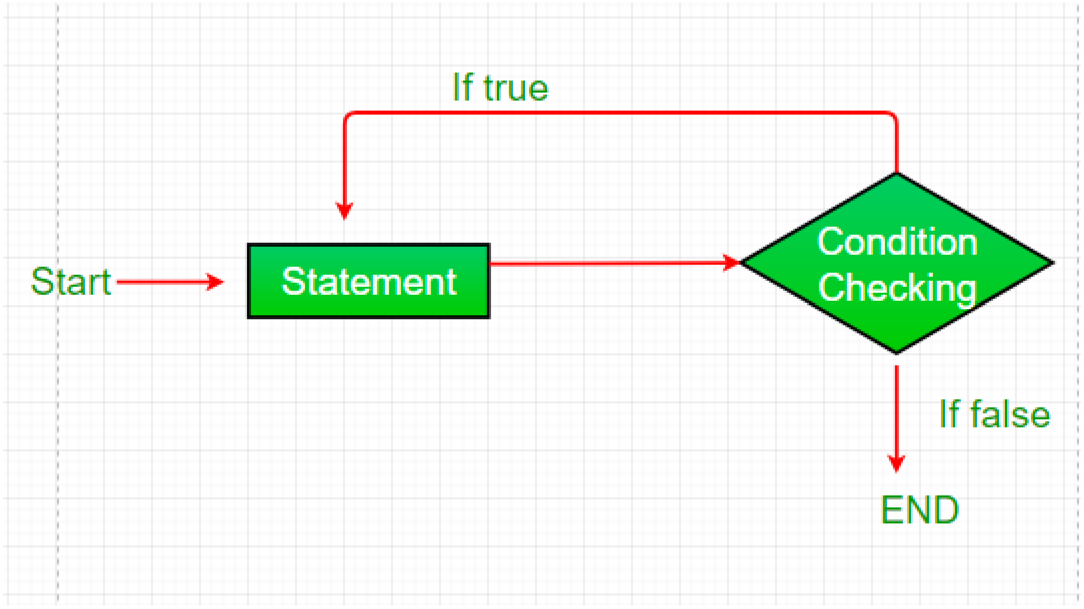
do while

- do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop.**

  Syntax:

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

Flowchart



- do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

```
•   class my
•   {
•       public static void main(String args[])
•       {
•           int x = 21;
•           do
•           {
•               // The line will be printed even
•               // if the condition is false
•               System.out.println("Value of x:" + x);
•               x++;
•           }
•           while (x < 20);
•       }
•   }
```

Output:

```
Value of x: 21
```

break statement

- By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
  class my
  {
   public static void main(String args[])
   {
        for (int i = 0; i < 100; i++)
         {
               if (i * 10 == 50)
               break; // terminate loop if i is 5
        System.out.println("i: " + i);
        }
        System.out.println("Loop completed.");
  }
}
```

- The ouput of the program is:

```
i: 0
i: 1
i: 2
i: 3
i: 4
Loop completed.
```

Example

- When used inside a set of nested loops, the break statement will only break out of the innermost loop.

```
Class my
{
public static void main(String args[])
{
 for (int i = 0; i < 3; i++)
 {
              System.out.print("Iteration " + i + ": ");
             for (int j = 0; j < 10; j++)
             {
              if (j == 5)
               break; // terminate loop if j is 5
             System.out.print(j + " ");
             }
              System.out.println();
}
              System.out.println("Loops completed.");
 }
}
```

Output:

```
 Iteration 0: 0 1 2 3 4
 Iteration 1: 0 1 2 3 4
 Iteration 2: 0 1 2 3 4
 Loops completed.
```

continue statement

- There may be situation in which you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.

- In other words, sometimes it is useful to force an early iteration of a loop. The continue statement performs such an action.

Example

```
class my
 {
        public static void main(String args[])
       {
                for (int i = 0; i < 10; i++)
                {
                        System.out.print(i + " ");
                        if (i % 2 == 0)
                                continue;
                        System.out.println(“ ");
                }
        }
}
```

Output

```
0 1
2 3
4 5
6 7
8 9
```

return statement

- The return statement exits from the current method, and control flow returns to where the method was invoked.

  Types:
- **return statement that returns a value**: To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

  eg: return ++count;

- **return statement that doesn't return a value**: The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

  eg: return;

Java program that returns expression

```
public class Program
 {
           static int computeSize(int height, int width)
          {
                         // Return an expression based on two arguments (variables).
                         return height * width;
          }
          public static void main(String[] args)
           {
                        // Assign to the result of computeSize.
                        int result = computeSize(10, 3);
                        System.out.println(result);
           }
 }
```

Output

```
 30
```

Java program that uses return statement, void method

```
public class Program
{
        static void displayPassword(String password)
        {
                    // Write the password to the console.
                   System.out.println("Password: " + password);
                   // Return if our password is long enough.
                   if (password.length() >= 5)
                    {
                    return;
                    }
                    System.out.println("Password too short!");
                   // An implicit return is here.
         }
         public static void main(String[] args)
         {
                    displayPassword("furball");
                    displayPassword("cat");
        }
}
```

Output

```
Password: furball
Password: cat
Password too short!
```