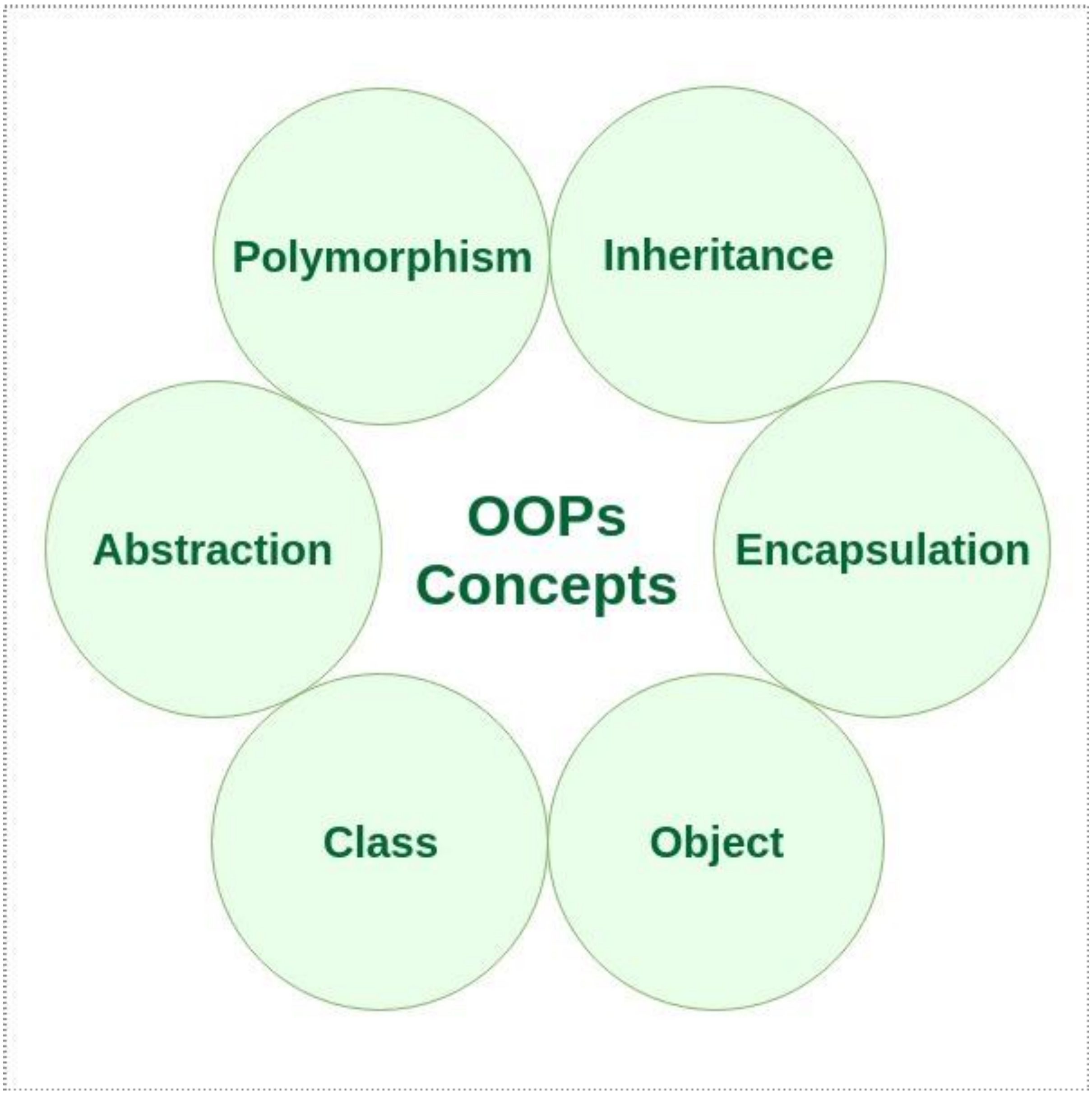
Access Modifier: Defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there are 4 types of access specifiers:

- **public:** Accessible in all classes in your application.
- **protected:** Accessible within the package in which it is defined and in its **subclass(es) (including subclasses declared outside the package)**.
- **private:** Accessible only within the class in which it is defined.
- **default (declared/defined without using any modifier):** Accessible within the same class and package within which its class is defined.

OOPS concepts are as follows:

1. Class
2. Object
3. Method and method passing
4. Pillars of OOPs
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism
        - Compile-time polymorphism
        - Runtime polymorphism



A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

1. **Modifiers**: A class can be public or have default access (Refer to this for details).
2. **Class name:** The class name should begin with the initial letter capitalized by convention.
3. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

4. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body is surrounded by braces, { }.

An object is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity**: It is a unique name given to an object that enables it to interact with other objects.
4. **Method**: A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping it, which is why they are considered **time savers**. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

class and objects one simple java program :

- Java

```
public class GFG {



    static String Employee_name;

    static float Employee_salary;



    static void set(String n, float p) {

        Employee_name  = n;

        Employee_salary  = p;

    }



    static void get() {

        System.out.println("Employee name is: " +Employee_name );
        System.out.println("Employee CTC is: " + Employee_salary);

    }


    public static void main(String args[]) {

        GFG.set("Rathod Avinash", 10000.0f);

        GFG.get();

    }

}
```

Output

```
Employee name is: Rathod Avinash

Employee CTC is: 10000.0
```

Let us now discuss the 4 pillars of OOPs:

Pillar 1: Abstraction

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or non-essential units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.
Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the object.
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the car speed or applying brakes will stop the car, but he does not know how on pressing the accelerator, the speed is actually increasing. He does not know about the inner mechanism of the car or the implementation of the accelerators, brakes etc. in the car. This is what abstraction is.
In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.
The abstract method contains only method declaration but not implementation.
Demonstration of Abstract class

- Java

```
//abstract class

abstract class GFG{

  //abstract methods declaration
  abstract void add();

  abstract void mul();

  abstract void div();

}
```

Pillar 2: Encapsulation

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates.

Encapsulation provides several benefits, including:

1. **Data hiding:** By hiding the implementation details of a class, encapsulation protects the data from unauthorized access and manipulation.
2. **Modularity:** Encapsulation helps to break down complex systems into smaller, more manageable components, making the codebase more modular and easier to maintain.
3. **Flexibility:** By providing a controlled interface for interacting with a class, encapsulation allows for changes to the internal implementation without affecting the external interface.

Demonstration of Encapsulation:

- Java

```
//Encapsulation using private modifier



//Employee class contains private data called employee id and employee name

class Employee {

    private int empid;

      private String ename;

}
```

Pillar 3: Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.
Let us discuss some frequently used important terminologies:

- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).
- **Subclass:** The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Demonstration of Inheritance :

- Java

```java
//base class or parent class or super class

class A{

  //parent class methods

  void method1(){}

  void method2(){}

}



//derived class or child class or base class

class B extends A{  //Inherits parent class methods

  //child class methods

  void method3(){}

  void method4(){}

}
```

Pillar 4: Polymorphism

It refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called **polymorphism**.

E.g.

- Java

```
sleep(1000) //millis


sleep(1000,2000) //millis,nanos
```

Note: *Polymorphism in Java is mainly of 2 types:*

1. *Overloading*
2. *Overriding*

Two more examples of polymorphism in Java are method overriding and method overloading.

In **method overriding**, the child class can use the OOP polymorphism concept to override a method of its parent class. That allows a programmer to use one method in different ways depending on whether it's invoked by an object of the parent class or an object of the child class.

In **method overloading,** a single method may perform different functions depending on the context in which it's called. This means a single method name might work in different ways depending on what arguments are passed to it.

Benefits of Polymorphism

Polymorphism provides several benefits, including:

1. **Flexibility:** Polymorphism allows for more flexible and adaptable code by enabling objects of different classes to be treated as if they are of the same class.
2. **Code reuse:** Polymorphism promotes code reuse by allowing classes to inherit functionality from other classes and to share common methods and properties.
3. **Simplification:** Polymorphism simplifies code by enabling the use of generic code that can handle different types of objects.

Polymorphism allows for more flexible and adaptable code. By enabling objects of different classes to be treated as if they are of the same class, polymorphism promotes code reuse, simplification, and flexibility, making it an essential component of Object-Oriented Programming.

Example

Java Buzzwords or Features of Java

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

1. Simple
2. Object-oriented
3. Distributed
4. Interpreted
5. Robust
6. Secure
7. Architecture neutral
8. Portable
9. High performance
10. Multithreaded
11. Dynamic

Java Buzzwords

1. Simple

- Java was designed to be easy for a professional programmer to learn and use effectively.
- It’s simple and easy to learn if you already know the basic concepts of Object Oriented Programming.
- Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

2. Object Oriented

- Java is a true object-oriented programming language.
- Almost the “Everything is an Object” paradigm. All program code and data reside within objects and classes.
- The object model in Java is simple and easy to extend.
- Java comes with an extensive set of classes, arranged in packages that can be used in our programs through inheritance.
- Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

The basic concepts of OOPs are:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

3. Distributed

- Java is designed to create distributed applications on networks.
- Java applications can access remote objects on the Internet as easily as they can do in the local system.
- Java enables multiple programmers at multiple remote locations to collaborate and work together on a single project.
- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.

4. Compiled and Interpreted

- Usually, a computer language is either compiled or Interpreted. Java combines both this approach and makes it a two-stage system.
- **Compiled:** Java enables the creation of cross-platform programs by compiling them into an intermediate representation called Java Bytecode.
- **Interpreted:** Bytecode is then interpreted, which generates machine code that can be directly executed by the machine that provides a Java Virtual machine.

5. Robust

- It provides many features that make the program execute reliably in a variety of environments.
- Java is a strictly typed language. It checks code both at compile time and runtime.
- Java takes care of all memory management problems with garbage collection.
- Java, with the help of exception handling, captures all types of serious errors and eliminates any risk of crashing the system.

6. Secure

- Java provides a “firewall” between a networked application and your computer.
- When a Java Compatible Web browser is used, downloading can be done safely without fear of viral infection or malicious intent.
- Java achieves this protection by confining a Java program to the Java execution environment and not allowing it to access other parts of the computer.

7. Architecture Neutral

- Java language and Java Virtual Machine helped in achieving the goal of "write once; run anywhere, any time, forever."
- Changes and upgrades in operating systems, processors and system resources will not force any changes in Java Programs.

8. Portable

- Java Provides a way to download programs dynamically to all the various types of platforms connected to the Internet.
- Java is portable because of the Java Virtual Machine (JVM). The JVM is an abstract computing machine that provides a runtime environment for Java programs to execute. The JVM provides a consistent environment for Java programs to run on, regardless of the underlying hardware and operating system. This means that a Java program can be written on one device and run on any other device with a JVM installed, without any changes or modifications.

9. High Performance

- Java performance is high because of the use of bytecode.
- The bytecode was used so that it can be easily translated into native machine code.

10. Multithreaded

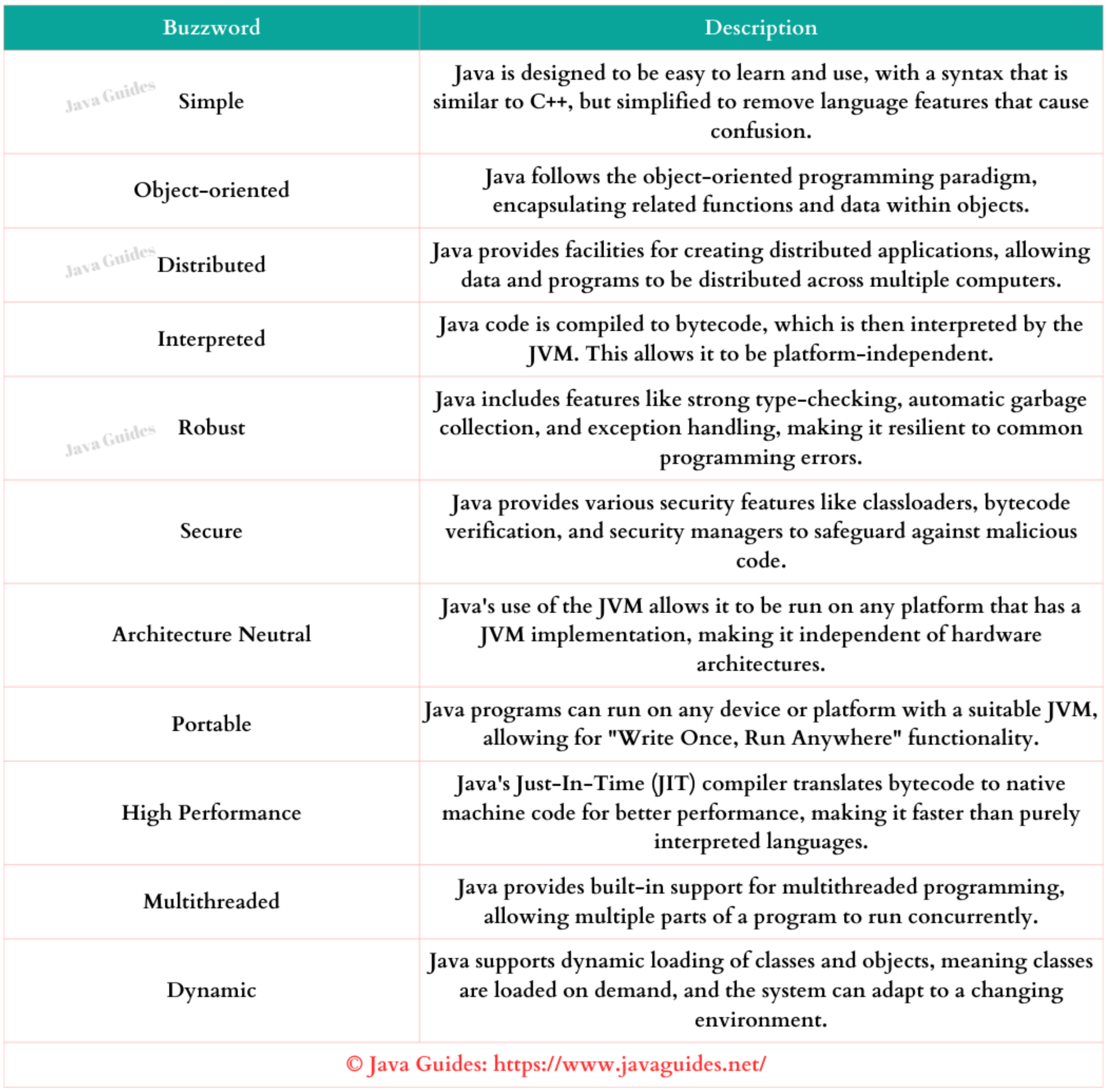
- Multithreaded Programs handled multiple tasks simultaneously, which was helpful in creating interactive, networked programs.

- Java run-time system comes with tools that support multiprocess synchronization used to construct smoothly interactive systems.

11. Dynamic

- Java is capable of linking in new class libraries, methods, and objects.
- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at runtime. This makes it possible to dynamically link code in a safe and expedient manner.

Java Buzzwords - Cheat Sheet



Buzzword	Description
Simple	Java is designed to be easy to learn and use, with a syntax that is similar to C++, but simplified to remove language features that cause confusion.
Object-oriented	Java follows the object-oriented programming paradigm, encapsulating related functions and data within objects.
Distributed	Java provides facilities for creating distributed applications, allowing data and programs to be distributed across multiple computers.
Interpreted	Java code is compiled to bytecode, which is then interpreted by the JVM. This allows it to be platform-independent.
Robust	Java includes features like strong type-checking, automatic garbage collection, and exception handling, making it resilient to common programming errors.
Secure	Java provides various security features like classloaders, bytecode verification, and security managers to safeguard against malicious code.
Architecture Neutral	Java's use of the JVM allows it to be run on any platform that has a JVM implementation, making it independent of hardware architectures.
Portable	Java programs can run on any device or platform with a suitable JVM, allowing for "Write Once, Run Anywhere" functionality.
High Performance	Java's Just-In-Time (JIT) compiler translates bytecode to native machine code for better performance, making it faster than purely interpreted languages.
Multithreaded	Java provides built-in support for multithreaded programming, allowing multiple parts of a program to run concurrently.
Dynamic	Java supports dynamic loading of classes and objects, meaning classes are loaded on demand, and the system can adapt to a changing environment.

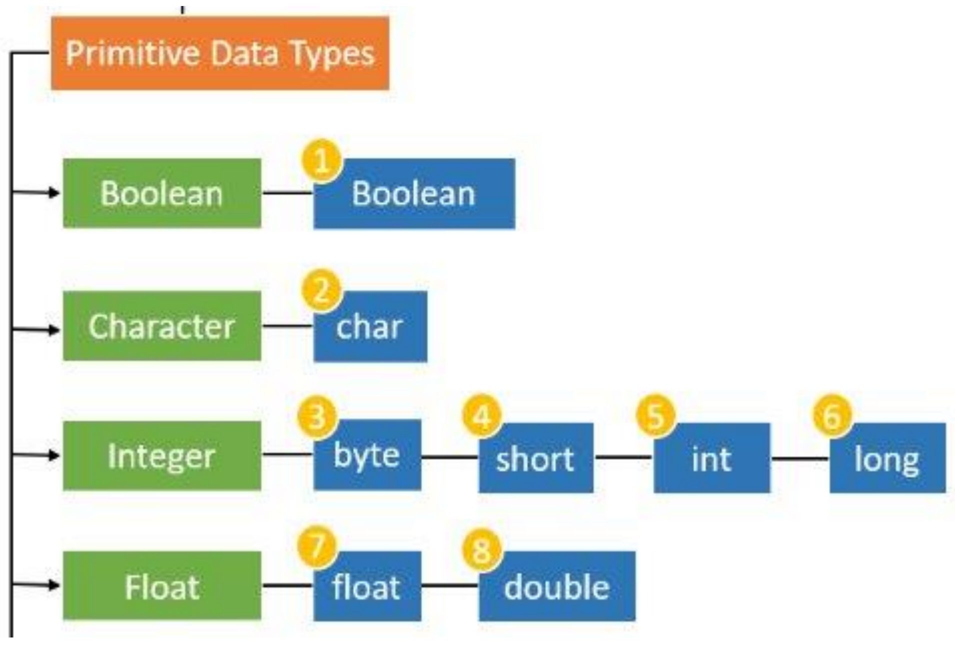
Data Types in Java

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

1. **Primitive Data Type:** such as boolean, char, int, short, byte, long, float, and double
2. **Non-Primitive Data Type or Object Data type:** such as String, Array, etc.

Primitive Data Types

Primitive data types specify the size and type of variable values. They are the building blocks of data manipulation and cannot be further divided into simpler data types.



Variables in Java

Java variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.
- In Java, all variables must be declared before use.

How to Initialize Variables in Java?

It can be perceived with the help of 3 components that are as follows:

- **datatype**: Type of data that can be stored in this variable.
- **variable_name**: Name given to the variable.
- **value**: It is the initial value stored in the variable.

```
Int age =20;
```

Types of Variables in Java

Now let us discuss different types of variables which are listed as follows:

1. Local Variables

2. Instance Variables
3. Static Variables

1. Local Variables

A variable defined within a block or method or constructor is called a local variable.

- These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
- Initialization of the local variable is mandatory before using it in the defined scope.

Time Complexity of the Method:

```
Time Complexity: O(1)
Auxiliary Space: O(1)
```

Below is the implementation of the above approach:

- Java

```
// Java Program to implement

// Local Variables

import java.io.*;


class GFG {

    public static void main(String[] args)

    {

        // Declared a Local Variable

        int var = 10;


        // This variable is local to this main method only

        System.out.println("Local Variable: " + var);

    }
}
```

Output

```
Local Variable: 10
```

Example :

- Java

```java
package a;

public class LocalVariable {

    public static void main(String[] args)

    {

        // x is a local variable

        int x = 10;


        // message is also a local

        // variable

        String message = "Hello, world!";


        System.out.println("x = " + x);

        System.out.println("message = " + message);


        if (x > 5) {

            // result is a

            // local variable

            String result = "x is greater than 5";

            System.out.println(result);

        }
        // Uncommenting the line below will result in a
        // compile-time error System.out.println(result);

        for (int i = 0; i < 3; i++) {
            String loopMessage
                = "Iteration "
                  + i; // loopMessage is a local variable
            System.out.println(loopMessage);
        }

        // Uncommenting the line below will result in a
        // compile-time error
        // System.out.println(loopMessage);
    }
}
```

Output :

```
message = Hello, world!
x is greater than 5
Iteration 0
Iteration 1
Iteration 2
```

2. Instance Variables

2. Instance Variables

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.

- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
- Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for Wrapper classes like *Integer* it is *null, etc.*
- Instance variables can be accessed only by creating objects.
- We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.

The complexity of the method:

```
Time Complexity: O(1)
Auxiliary Space: O(1)
```

Below is the implementation of the above approach:

- Java

```java
// Java Program to demonstrate

// Instance Variables

import java.io.*;



class GFG {



    // Declared Instance Variable

    public String geek;

    public int i;

    public Integer I;

    public GFG()

    {

        // Default Constructor

        // initializing Instance Variable

        this.geek = "Shubham Jain";
    }


    // Main Method
    public static void main(String[] args)
    {
        // Object Creation
        GFG name = new GFG();


        // Displaying O/P
        System.out.println("Geek name is: " + name.geek);
        System.out.println("Default value for int is "
                           + name.i);


        // toString() called internally
        System.out.println("Default value for Integer is "
                           + name.I);
    }
}
```

Output

```
Geek name is: Shubham Jain
Default value for int is 0
Default value for Integer is null
```

3. Static Variables

Static variables are also known as class variables.

- These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.

- Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For *String* it is *null*, for *float* it is *0.0f*, for *int* it is *0*, for *Wrapper classes* like *Integer* it is *null*, etc.
- If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.
- Static variables cannot be declared locally inside an instance method.
- Static blocks can be used to initialize static variables.

The complexity of the method:

```
Time Complexity: O(1)
Auxiliary Space: O(1)
```

Below is the implementation of the above approach:

- Java

```java
// Java Program to demonstrate

// Static variables

import java.io.*;



class GFG {

    // Declared static variable

    public static String geek = "Shubham Jain";



    public static void main(String[] args)
    {

        // geek variable can be accessed without object
        // creation Displaying O/P GFG.geek --> using the
        // static variable
        System.out.println("Geek Name is : " + GFG.geek);

        // static int c=0;
        // above line,when uncommented,
        // will throw an error as static variables cannot be
        // declared locally.
    }
}
```

Output

```
Geek Name is : Shubham Jain
```

Differences Between the Instance Variables and the Static Variables

Now let us discuss the differences between the Instance variables and the Static variables:

- Each object will have its own copy of an instance variable, whereas we can only have one copy of a static variable per class, irrespective of how many objects we create. Thus, **static variables** are good for **memory management**.
- Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of a static variable, changes will be reflected in other objects as static variables are common to all objects of a class.
- We can access instance variables through object references, and static variables can be accessed directly using the class name**.**

- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed. Static variables are created when the program starts and destroyed when the program stops.

Java array

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

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

Advantgaes:

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)

2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

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

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

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

Test it Now

Output:

```
10
20
70
40
50
```

Multidimensional Array in Java

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

Syntax to Declare Multidimensional Array in Java

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

Example to instantiate Multidimensional Array in Java

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

Example to initialize Multidimensional Array in Java

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

Example of Multidimensional Java Array

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

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
```

12. }
13. }}

Test it Now

Output:

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

Types of Operators in Java

There are multiple types of operators in Java all are mentioned below:

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators
9. instance of operator

1. Arithmetic Operators

They are used to perform simple arithmetic operations on primitive data types.

- *** :** Multiplication
- **/ :** Division
- **% :** Modulo
- **+ :** Addition
- **– :** Subtraction

Example:

Java

```
// Java Program to implement
// Arithmetic Operators
import java.io.*;

// Drive Class
class GFG {
      // Main Function
    public static void main (String[] args) {

        // Arithmetic operators
        int a = 10;
        int b = 3;
        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));

    }
}
```

Output

```
a + b = 13

a - b = 7

a * b = 30

a / b = 3

a % b = 1
```

2. Unary Operators

Unary operators need only one operand. They are used to increment, decrement, or negate a value.

- **– : Unary minus**, used for negating the values.
- **+ : Unary plus** indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
- **++ : Increment operator**, used for incrementing the value by 1. There are two varieties of increment operators.

    - **Post-Increment:** Value is first used for computing the result and then incremented.
    - **Pre-Increment:** Value is incremented first, and then the result is computed.
- **– – : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operators.

    - **Post-decrement:** Value is first used for computing the result and then decremented.

    - **Pre-Decrement: The value** is decremented first, and then the result is computed.
- **! : Logical not operator**, used for inverting a boolean value.

Example:
Java

```java
// Java Program to implement
// Uniary Operators
import java.io.*;

// Driver Class
class GFG {
      // main function
    public static void main(String[] args)
    {
        // Interger declared
        int a = 10;
        int b = 10;

        // Using unary operators
        System.out.println("Postincrement : " + (a++));
        System.out.println("Preincrement : " + (++a));

        System.out.println("Postdecrement : " + (b--));
        System.out.println("Predecrement : " + (--b));
    }
}
```

Output

```
Postincrement : 10

Preincrement : 12

Postdecrement : 10

Predecrement : 8
```

3. Assignment Operator

'=' Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is

assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.
The general format of the assignment operator is:

```
variable = value;
```

In many cases, the assignment operator can be combined with other operators to build a shorter version of the statement called a **Compound Statement**. For example, instead of a **=** a+5, we can write a **+=** 5.

- **+=**, for adding the left operand with the right operand and then assigning it to the variable on the left.
- **-=**, for subtracting the right operand from the left operand and then assigning it to the variable on the left.
- ***=**, for multiplying the left operand with the right operand and then assigning it to the variable on the left.
- **/=**, for dividing the left operand by the right operand and then assigning it to the variable on the left.
- **%=**, for assigning the modulo of the left operand by the right operand and then assigning it to the variable on the left.

Example:
Java

```java
// Java Program to implement
// Assignment Operators
import java.io.*;

// Driver Class
class GFG {
    // Main Function
    public static void main(String[] args)
    {

        // Assignment operators
        int f = 7;
        System.out.println("f += 3: " + (f += 3));
        System.out.println("f -= 2: " + (f -= 2));
        System.out.println("f *= 4: " + (f *= 4));
        System.out.println("f /= 3: " + (f /= 3));
        System.out.println("f %= 2: " + (f %= 2));
        System.out.println("f &= 0b1010: " + (f &= 0b1010));
        System.out.println("f |= 0b1100: " + (f |= 0b1100));
        System.out.println("f ^= 0b1010: " + (f ^= 0b1010));
        System.out.println("f <<= 2: " + (f <<= 2));
        System.out.println("f >>= 1: " + (f >>= 1));
        System.out.println("f >>>= 1: " + (f >>>= 1));
    }
}
```

Output

```
f += 3: 10
```

```
f -= 2: 8

f *= 4: 32

f /= 3: 10

f %= 2: 0

f &= 0b1010: 0

f |= 0b1100: 12

f ^= 0b1010: 6

f <<= 2: 24

f >>= 1: 12

f >>>= 1: 6
```

4. Relational Operators

These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is,

```
variable relation_operator value
```

Some of the relational operators are-

- **==, Equal to** returns true if the left-hand side is equal to the right-hand side.
- **!=, Not Equal to** returns true if the left-hand side is not equal to the right-hand side.
- **<, less than:** returns true if the left-hand side is less than the right-hand side.
- **<=, less than or equal to** returns true if the left-hand side is less than or equal to the right-hand side.
- **>, Greater than:** returns true if the left-hand side is greater than the right-hand side.
- **>=, Greater than or equal to** returns true if the left-hand side is greater than or equal to the right-hand side.

Example:

Java

```
// Java Program to implement
```

```java
// Relational Operators
import java.io.*;

// Driver Class
class GFG {
    // main function
    public static void main(String[] args)
    {
        // Comparison operators
        int a = 10;
        int b = 3;
        int c = 5;

        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 == c: " + (a == c));
        System.out.println("a != c: " + (a != c));
    }
}
```

Output

```
a > b: true

a < b: false

a >= b: true

a <= b: false

a == c: false

a != c: true
```

5. Logical Operators

These operators are used to perform "logical AND" and "logical OR" operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has "Logical NOT", which returns true when the condition is false and vice-versa

Conditional operators are:

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.
- **!, Logical NOT:** returns true when a condition is false and vice-versa

Example:

Java

```
// Java Program to implemenet
// Logical operators
import java.io.*;

// Driver Class
class GFG {
      // Main Function
    public static void main (String[] args) {
        // Logical operators
        boolean x = true;
        boolean y = false;

        System.out.println("x && y: " + (x && y));
        System.out.println("x || y: " + (x || y));
        System.out.println("!x: " + (!x));
    }
}
```

Output

```
x && y: false

x || y: true

!x: false
```

6. Ternary operator

The ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary.

The general format is:

```
condition ? if true : if false
```

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

Example:

Java

```
// Java program to illustrate
// max of three numbers using
// ternary operator.
public class operators {
     public static void main(String[] args)
     {
          int a = 20, b = 10, c = 30, result;

          // result holds max of three
          // numbers
          result
               = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
          System.out.println("Max of three numbers = "
                             + result);
     }
}
```

Output

```
Max of three numbers = 30
```

7. Bitwise Operators

These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit-by-bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's complement representation of the input value, i.e., with all bits inverted.

Java

```
// Java Program to implement
// bitwise operators
import java.io.*;

// Driver class
class GFG {
     // main function
     public static void main(String[] args)
     {
         // Bitwise operators
         int d = 0b1010;
         int e = 0b1100;
         System.out.println("d & e: " + (d & e));
         System.out.println("d | e: " + (d | e));
         System.out.println("d ^ e: " + (d ^ e));
         System.out.println("~d: " + (~d));
         System.out.println("d << 2: " + (d << 2));
         System.out.println("e >> 1: " + (e >> 1));
         System.out.println("e >>> 1: " + (e >>> 1));
     }
}
```

Output

```
d & e: 8

d | e: 14

d ^ e: 6

~d: -11

d << 2: 40

e >> 1: 6

e >>> 1: 6
```

8. Shift Operators

These operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two. General format-

```
 number shift_op number_of_places_to_shift;
```

- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as multiplying the number with some power of two.
- **>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the

sign of the initial number. Similar effect to dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

Java

```
// Java Program to implement
// shift operators
import java.io.*;

// Driver Class
class GFG {
    // main function
    public static void main(String[] args)
    {
        int a = 10;

          // using left shift
        System.out.println("a<<1 : " + (a << 1));

        // using right shift
        System.out.println("a>>1 : " + (a >> 1));
    }
}
```

Output

```
a<<1 : 20

a>>1 : 5
```

Advantages of Operators in Java

The advantages of using operators in Java are mentioned below:

1. **Expressiveness**: Operators in Java provide a concise and readable way to perform complex calculations and logical operations.
2. **Time-Saving:** Operators in Java save time by reducing the amount of code required to perform certain tasks.
3. **Improved Performance**: Using operators can improve performance because they are often implemented at the hardware level, making them faster than equivalent Java code.

Disadvantages of Operators in Java

The disadvantages of Operators in Java are mentioned below:

1. **Operator Precedence:** Operators in Java have a defined precedence, which can lead to unexpected results if not used properly.
2. **Type Coercion**: Java performs implicit type conversions when using operators, which can lead to unexpected results or errors if not used properly.

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

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

Decision-Making statements:

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

1) If Statement:

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

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

Let's understand the if-statements one by one.

1) Simple if statement:

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

Syntax of if statement is given below.

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

Consider the following example in which we have used the **if** statement in the java code.

Student.java

Student.java

```
public class Student {

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

Output:

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

2) if-else statement

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

Syntax:

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

Consider the following example.

Student.java

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

Output:

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

3) if-else-if ladder:

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

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

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

Consider the following example.

Student.java

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

Output:

```
Delhi
```

4. Nested if-statement

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

Syntax of Nested if-statement is given below.

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

Consider the following example.

Student.java

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

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

Output:

```
Delhi
```

Switch Statement:

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

Points to be noted about switch statement:

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

The syntax to use the switch statement is given below.

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

Consider the following example to understand the flow of the switch statement.

Student.java

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

Output:

```
2
```

Loop Statements

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

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

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

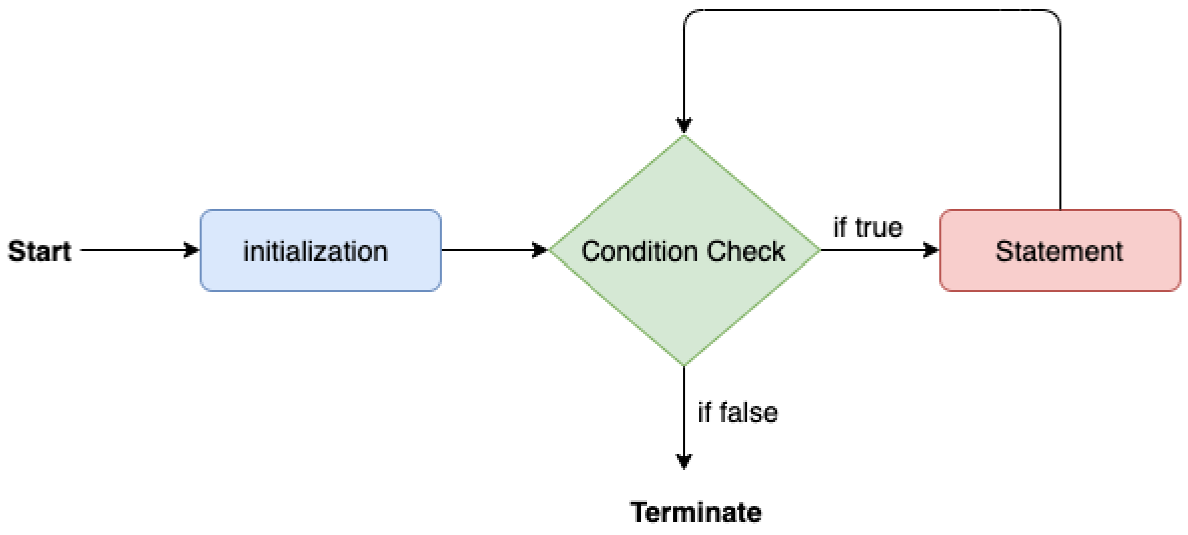
Let's understand the loop statements one by one.

Java for loop

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

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

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



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

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

Output:

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

Java for-each loop

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

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

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

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

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

Java while loop

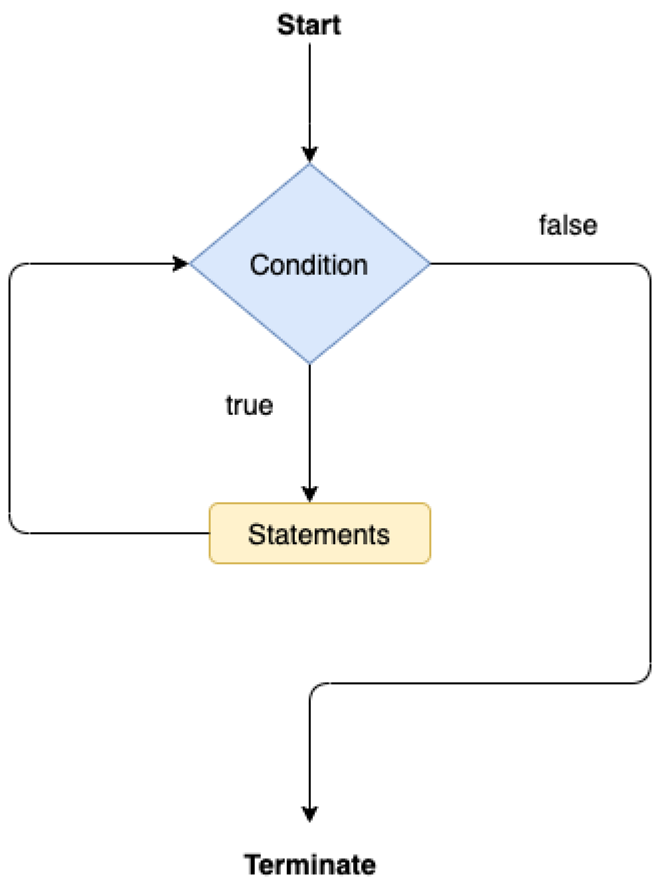
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while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

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

The syntax of the while loop is given below.

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

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



Consider the following example.

Calculation .java

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

```java
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

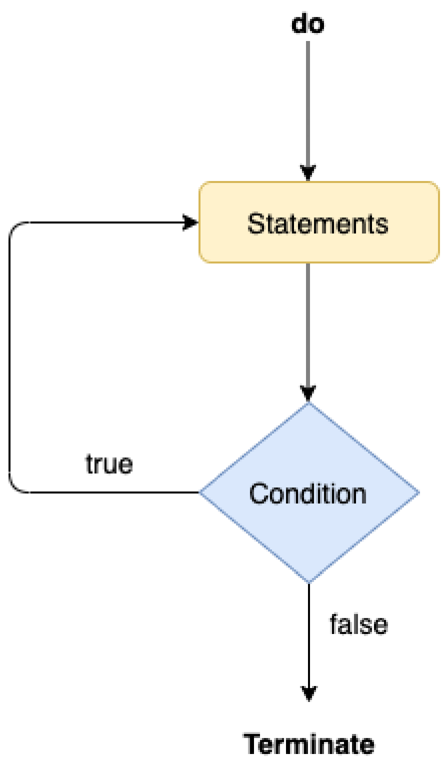
Java do-while loop

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

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

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

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



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

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

Output:

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

```
10
```

Jump Statements

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

Java break statement

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

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

The break statement example with for loop

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

BreakExample.java

```java
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

```
0
1
2
3
```

```
4
5
6
```

Java continue statement

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

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

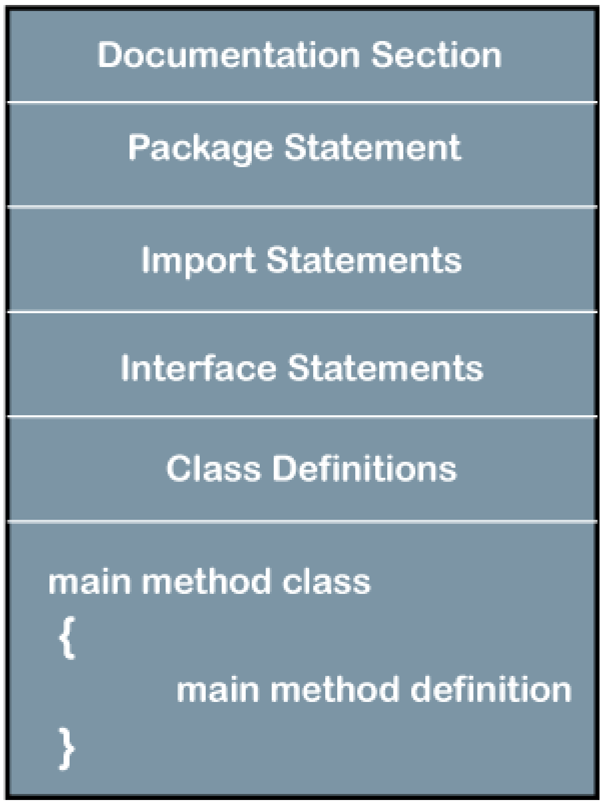
```java
public class ContinueExample {

public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {

if(j == 4) {
continue;
}
System.out.println(j);
}
}
}

}
```

Output:

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

Structure of Java Program

Java is an object-oriented programming, **platform-independent,** and **secure** programming language that makes it popular.



Structure of Java Program

1. **Documentation Section**

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line,** and **documentation** comments.

2. **Package statement:-** The package statement identifies the package that the class belongs to. We put classes into sensible groups with the help of packages.

   ```
   package javatpoint; //where javatpoint is the package name
   package com.javatpoint; //where com is the root directory and javatpoint is th
   e subdirectory
   ```

3. **Import statements:** The import statements import other classes into the current class. This allows the current class to use the methods and variables of the imported classes.

   ```
   import java.util.Scanner; //it imports the Scanner class only
   import java.util.*; //it imports all the class of the java.util package
   ```

4. **Class definition:** The class definition defines the class. A class definition is composed of –
   1. class name
   2. class variables
   3. methods
   4. constructors
5. **Main method:** The Java program starts with the main method.
6. **Body of the class:-** The body of the class contains the code for the class. The code inside a class can be divided into methods, constructors, and variables.

Java Classes

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.

4. A Class in Java can contain:
    - Data member
    - Method
    - Constructor
    - Nested Class
    - Interface

Components of Java Classes

In general, class declarations can include these components, in order:

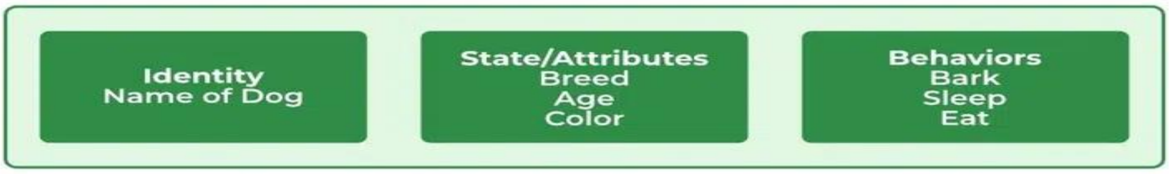
1. ***Modifiers****: A class can be public or has default access (Refer this for details).*
2. ***Class keyword:*** *class keyword is used to create a class.*
3. ***Class name:*** *The name should begin with an initial letter (capitalized by convention).*
4. ***Superclass(if any):*** *The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.*
5. ***Interfaces(if any):*** *A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.*
6. ***Body:*** *The class body is surrounded by braces, { }.*

Java Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog


Identity
Name of Dog
State/Attributes
Breed
Age
Color
Behaviors
Bark
Sleep
Eat