* **Package:** A package groups related classes, interfaces, and sub-packages. To use pre-defined classes from other packages, a programmer needs to import those packages. Java automatically imports the **java.lang.\*** package by default.
* **Class:**A class is a user-defined data type. Every Java program starts with at least one class definition. If we declare a class as**public**, it means it is accessible from another package.
* **Class name:** It is the name given to that class; it acts like a new data type that we can use to create objects.
* **Data members:** Data members are variables declared inside a class.
* **Constructor:**A constructor is a special block of code that runs automatically when we create a new object from a class. The main task of a constructor is to initialize the new object. Always remember the constructor name must be the same as the class name, also it does not have any return type. .

package list;

public class ClassName {

// Data members

// Constructor functions

// User-defined methods

public static void main(String[] arguments) {

// Block of statements

}

}

**User-Defined Methods**

User-defined methods in Java can be instance or static and contain the core logic of the program. The main() method is the starting point of every Java program, must be public static void, and takes a String[] as an argument.

**Types of Data Members**

Java Class is a collection of data members and functions. Any Java program may contain two types of data members.

* Instance or non-static data members
* Static or class data members

**Instance Data Members vs Static Data Members**

The table below demonstrates the difference between Instance data member and Static data memeber.

| **Instance Data Members** | **Static Data Members** |
| --- | --- |
| Memory is allocated every time an object is created. | Memory is allocated only once when the class is loaded. |
| Each object has its own copy of instance variables. | All objects share a single copy of static variables |
| Declared without the static keyword | Declared using the static keyword. |
| Accessed using the object name | Accessed using the class name |
| It is known as Object-level data members. | It is Known as Class-level data members. |
| Value can be different for each object. | Value is common for all objects of the class. |

**Types of Methods**

In Java programs generally, we may define two types of methods apart from the constructor.

* Instance or non-static methods
* Static or class methods

**Instance Methods vs Static Methods**

The table below demonstrates the difference between Instance methods and Static methods.

| **Instance Methods** | **Static Methods** |
| --- | --- |
| Used to perform tasks related to individual objects, such as reading records from a file or processing object-specific data. | Used to perform operations that are related to the class as a whole, such as utility functions or managing shared resources. |
| Do not require the static keyword in their definition.  Syntax: void net\_salary(parameters) { statements; } | Must include the static keyword in their definition.  Syntax: static void basic\_salary(parameters) { statements; } |
| Accessed through an object of the class | Accessed through the class name. |
| Can access both instance variables and static variable | Cannot access instance variables or methods without an object reference. |

# Accessing Protected Members in Java

In Java, there are four types of access modifiers. These are public, private, default, and protected. To get the idea of these modifiers, you can refer to In Java, there are four types of access modifiers. These are public, private, default, and protected. To get the idea of these modifiers, you can refer to [access modifiers in java](https://www.geeksforgeeks.org/access-modifiers-java/). In this article, we discuss the accessibility of protected members in different cases.

Now let us discuss various scenarios of accessing protected members which are listed below as follows:

1. Accessing in the same class
2. Accessing in other classes of the same package
3. Accessing protected members of a class in its subclass in the same package
4. Accessing another class in a different package
5. Accessing in sub-class in a different package

**Case 1:** Accessing protected members in the same class

*We can access protected members of a class anywhere in it.*

n this article, we discuss the accessibility of protected members in different cases.

Now let us discuss various scenarios of accessing protected members which are listed below as follows:

1. Accessing in the same class
2. Accessing in other classes of the same package
3. Accessing protected members of a class in its subclass in the same package
4. Accessing another class in a different package
5. Accessing in sub-class in a different package

**Case 1:** Accessing protected members in the same class

*We can access protected members of a class anywhere in it.*

*// Java Program to Illustrate*

*// Accessing Protected Members*

*// in the same class*

*// Main class*

*class Sample {*

*protected int year = 2021;*

*protected void printYear()*

*{*

*System.out.println("Its " + year + " !!");*

*}*

*public static void main(String[] args)*

*{*

*Sample sample = new Sample();*

*System.out.println(sample.year);*

*sample.printYear();*

*}*

*}*

| **ype** | **Description** | **Default** | **Size** | **Example Literals** | **Range of values** |
| --- | --- | --- | --- | --- | --- |
| **boolean** | true or false | false | JVM-dependent (typically 1 byte) | true, false | true, false |
| **byte** | 8-bit signed integer | 0 | 1 byte | (none) | -128 to 127 |
| **char** | Unicode character(16 bit) | \u0000 | 2 bytes | 'a', '\u0041', '\101', '\\', '\', '\n', 'β' | 0 to 65,535 (unsigned) |
| **short** | 16-bit signed integer | 0 | 2 bytes | (none) | -32,768 to 32,767 |
| **int** | 32-bit signed integer | 0 | 4 bytes | -2,0,1 | -2,147,483,648  to  2,147,483,647 |
| **long** | 64-bit signed integer | 0L | 8 bytes | -2L,0L,1L | -9,223,372,036,854,775,808  to  9,223,372,036,854,775,807 |
| **float** | 32-bit IEEE 754 floating-point | 0.0f | 4 bytes | 3.14f, -1.23e-10f | ~6-7 significant decimal digits |
| **double** | 64-bit IEEE 754 floating-point | 0.0d | 8 bytes | 3.1415d, 1.23e100d | ~15-16 significant decimal digits |

# Scope of Variables in Java

The **scope of variables** is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e., scope of a variable can be determined at compile time and independent of the function call stack.

**Java Scope of Variables**

Java Scope Rules can be covered under the following categories.

* Instance Variables
* Static Variables
* Local Variables
* Parameter Scope
* Block Scope

### **. 1. Instance Variables - Class Level Scope**

These variables must be declared inside a class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

*public class Test {  
// All variables defined directly inside a class  
// are member variables  
int a;  
private String b;  
  
void method1() {....}   
int method2() {....}  
  
char c;   
}*

* We can declare class variables anywhere in the class, but outside methods.
* Access specified of member variables doesn't affect scope of them within a class.
* Member variables can be accessed outside a class.

#### Access Modifiers and Member Variable Scope

| **Modifier** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- |
| **public** | Yes | Yes | Yes |
| **protected** | Yes | Yes | No |
| **Default (no modifier)** | Yes | No | No |
| **private** | No | No | No |

### 2. Static Variables - Class Level Scope

Static Variable is a type of class variable shared across instances. Static Variables are the variables which once declared can be used anywhere even outside the class without initializing the class. Unlike Local variables it scope is not limited to the class or the block.

import java.io.\*;

class Test{

// static variable in Test class

static int var = 10;

}

class Geeks

{

public static void main (String[] args) {

// accessing the static variable

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

}

}

### 3. Method Level Scope - Local Variable

Variables declared inside a method have method level scope and can't be accessed outside the method.

*public class Test {  
void method1()  
{  
// Local variable (Method level scope)  
int x;  
}  
}*

### 4. Parameter Scope - Local Variable

Here's another example of method scope, except this time the variable got passed in as a parameter to the method:

*class Test {  
private int x;  
  
public void setX(int x) {  
this.x = x;  
}  
}*

public class Geeks

{

// Class Scope variable

static int x = 11;

// Instance Variable

private int y = 33;

// Parameter Scope (x)

public void testFunc(int x) {

// Method Scope (t)

Geeks t = new Geeks();

this.x = 22;

y = 44;

// Printing variables with different scopes

System.out.println("Geeks.x: " + Geeks.x);

System.out.println("t.x: " + t.x);

System.out.println("t.y: " + t.y);

System.out.println("y: " + y);

}

// Main Method

public static void main(String args[]) {

Geeks t = new Geeks();

t.testFunc(5);

}

}

Geeks.x: 22

t.x: 22

t.y: 33

y: 44

### **Block Level Scope**

A variable declared inside pair of brackets "{" and "}" in a method has scope within the brackets only.

public class Test

{

public static void main(String args[])

{

// Block Level Scope

{

// The variable x has scope within

// brackets

int x = 10;

System.out.println(x);

}

// Uncommenting below line would produce

// error since variable x is out of scope.

// System.out.println(x);

}

}

Output

10

# Java - symbolic constants

In Java, a symbolic constant is a named constant value defined once and used throughout a program. Symbolic constants are declared using the final keyword.

* Which indicates that the value cannot be changed once it is initialized.
* The naming convention for symbolic constants is to use all capital letters with underscores separating words.

**Syntax of Symbolic Constants**

final data\_type CONSTANT\_NAME = value;

* **final**: The final keyword indicates that the value of the constant cannot be changed once it is initialized.
* **data\_type**: The data type of the constant such as int, double, boolean, or String.
* **CONSTANT\_NAME**: The name of the constant which should be written in all capital letters with underscores separating words.
* **value:** The initial value of the constant must be of the same data type as the constant.

import java.io.\*;

public class Constant {

public static final int MAX\_SIZE = 10;

public static void main(String[] args)

{

int[] array = new int[MAX\_SIZE];

for (int i = 0; i < MAX\_SIZE; i++) {

array[i] = i \* 2;

}

System.out.print("Array: ");

for (int i = 0; i < MAX\_SIZE; i++) {

System.out.print(array[i] + " ");

}

System.out.println();

}

}