What is Java?

Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts), class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.

Java Example

Let's have a quick look at Java programming example. A detailed description of Hello Java example is available in next page.

1. **class** Simple{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Hello Java");
4. }
5. }

**Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

## Java Example

Let's have a quick look at Java programming example. A detailed description of Hello Java example is available in next page.

1. **class** Simple{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Hello Java");
4. }
5. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=Simple)

## Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

## Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

#### **1) Standalone Application**

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

#### **2) Web Application**

An application that runs on the server side and creates a dynamic page is called a web application. Currently, [Servlet](https://www.javatpoint.com/servlet-tutorial), [JSP](https://www.javatpoint.com/jsp-tutorial), [Struts](https://www.javatpoint.com/struts-2-tutorial), [Spring](https://www.javatpoint.com/spring-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. technologies are used for creating web applications in Java.

#### **3) Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called enterprise application. It has advantages of the high-level security, load balancing, and clustering. In Java, [EJB](https://www.javatpoint.com/ejb-tutorial) is used for creating enterprise applications.

#### **4) Mobile Application**

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

# **Features of Java**

The primary objective of [Java programming](https://www.javatpoint.com/java-tutorial) language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java buzzwords.

A list of most important features of Java language is given below.



1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

### **Object-oriented**

Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. [Object](https://www.javatpoint.com/object-and-class-in-java)
2. Class
3. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
4. [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
5. [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
6. [Encapsulation](https://www.javatpoint.com/encapsulation)

**Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.

**Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.

## What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



## What happens at runtime?

At runtime, following steps are performed:



|  |
| --- |
| **Classloader:**is the subsystem of JVM that is used to load class files. |
| **Bytecode Verifier:**checks the code fragments for illegal code that can violate access right to objects. |
| **Interpreter:**read bytecode stream then execute the instructions. |

**Date Types**

--------------

**Rule of Using package and improt in Java**

package xyzpackage;

import abcpackage.\*;

public class Test {

}

### **Understanding Java Access Modifiers**

in case of the object oriented languages, the data and the operational logic in functions is bundled together as an object. This creates a way for **code reusability** and **security,**which are the two main concepts of encapsulation.

We use access modifiers to define access control for classes, methods and variables.

Error when we break access modifier rules;

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The method addTwoNumbers(int, int) from the type Addition is not visible

at xyzpackage.Test.main(Test.java:12)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### **1) Private**

The private access modifier is accessible only within the class.

**Simple example of private access modifier**

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### **Role of Private Constructor**

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### **Note: A class cannot be private or protected except nested class.**

### **2) Default**

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

1. //save by A.java
2. **package** pack;
3. **class** A{
4. **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9. **class** B{
10. **public** **static** **void** main(String args[]){
11. A obj = **new** A();//Compile Time Error
12. obj.msg();//Compile Time Error
13. }
14. }

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### **3) Protected**

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **protected** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B **extends** A{
11. **public** **static** **void** main(String args[]){
12. B obj = **new** B();
13. obj.msg();
14. }
15. }

Output:Hello

### **4) Public**

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Example of public access modifier**

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

### **Java Access Modifiers with Method Overriding**

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

1. **class** A{
2. **protected** **void** msg(){System.out.println("Hello java");}
3. }
5. **public** **class** Simple **extends** A{
6. **void** msg(){System.out.println("Hello java");}//C.T.Error
7. **public** **static** **void** main(String args[]){
8. Simple obj=**new** Simple();
9. obj.msg();
10. }
11. }

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Parameter Working in Java

A *parameter*is a value that you can pass to a method in Java. Then the method can use the parameter as though it were a local variable initialized with the value of the variable passed to it by the calling method.

The guessing-game application has a method named getRandomNumber that returns a random number between 1 and 10:

public static int getRandomNumber()

{

return (int)(Math.random() \* 10) + 1;

}

This method is useful, but it would be even more useful if you could tell it the range of numbers you want the random number to fall in. It would be nice to call the method like this to get a random number between 1 and 10:

int number = getRandomNumber(1, 10);

Then, if your program needs to roll dice, you could call the same method:

int number = getRandomNumber(1, 6);

Or, to pick a random card from a deck of 52 cards, you could call it like this:

int number = getRandomNumber(1, 52);

You wouldn’t have to start with 1, either. To get a random number between 50 and 100, you’d call the method like this:

## Declaring parameters

A method that accepts parameters must list the parameters in the method declaration. The parameters are placed in a *parameter list*inside the parentheses that follow the method name. For each parameter used by the method, you list the parameter type followed by the parameter name. If you need more than one parameter, you separate the parameters with commas.

Here’s a version of the getRandomNumber method that accepts parameters:

public static int getRandomNumber(int min, int max)

{

return (int)(Math.random()

\* (max – min + 1)) + min;

}

Here the method uses two parameters, both of type int, named min and max. Then, within the body of the method, these parameters can be used as though they were local variables.

int min = 1;

int max = 10;

int number = getRandomNumber(min, max);

## Understanding pass-by-value

When Java passes a variable to a method via a parameter, the method itself receives a copy of the variable’s value, not the variable itself. This copy is called a *pass-by-value,* and it has an important consequence: If a method changes the value it receives as a parameter, that change is *not*reflected in the original variable that was passed to the method. This program can help clear this up:

public class ChangeParameters

{

public static void main(String[] args)

{

int number = 1;

tryToChangeNumber(number);

System.out.println(number);

}

public static void tryToChangeNumber(int i)

{

i = 2;

}

}

Here a variable named number is set to 1 and then passed to the method named tryToChangeNumber. This method receives the variable as a parameter named i and then sets the value of i to 2. Meanwhile, back in the main method, println is used to print the value of number after the tryToChangeNumber method returns.

Because tryToChangeNumber gets only a copy of number, not the number variable itself, this program displays the following on the console: 1.

The key point is this: Even though the tryToChangeNumber method changes the value of its parameter, that change has no effect on the original variable that was passed to the method.

### **Coupling**

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

### **Cohesion**

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

**Why we are using getter and setter?**

By using getter and setter, the programmer can control how their important variables are accessed and updated, such as changing value of a variable within a specified range. Consider the following code of a setter method:

By using getter and setter we using the encapsulation concept of OOPs system where user decide what data must be show at what level.

# Getters & Setters

Getters and Setters are used to effectively protect your data, particularly when creating classes. For each instance variable, a getter method returns its value while a setter method sets or updates its value. Getters and setters are also known as accessors and mutators, respectively.

By convention, getters start with get, followed by the variable name, with the first letter of the variable name capitalized. Setters start with set, followed by the variable name, with the first letter of the variable name capitalized.

**Example:**

public class Vehicle {

private String color;

// Getter

public String getColor() {

return color;

}

// Setter

public void setColor(String c) {

this.color = c;

}

}