Java

What is Java?

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

## 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 an enterprise application. It has advantages like 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:

### **Simple**

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

### **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 incorporate both data and behavior.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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](https://www.javatpoint.com/object-and-class-in-java#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)

### **Platform Independent**



Java is platform independent because it is different from other languages like [C](https://www.javatpoint.com/c-programming-language-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

### **Secured**

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

* **No explicit pointer**
* **Java Programs run inside a virtual machine sandbox**



* **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 rights to objects.
* **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

### **Robust**

The English mining of Robust is strong. Java is robust because:

* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

### **Architecture-neutral**

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

### **Portable**

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

### **High-performance**

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

### **Distributed**

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

### **Multi-threaded**

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

### **Dynamic**

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

### **Difference between JDK and JRE**

**JRE** : The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.



**JDK** : The JDK also called Java Development Kit is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.



JVM:

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your \*.java files to obtain \*.class files that contain the bytecodes understandable by the JVM.

JVM control execution of every Java program.

## JVM Architecture



**Class Loader :** Class loader loads the Class for execution.

**Method area :** Stores pre-class structure as constant pool.

**Heap :** Heap is a memory area in which objects are allocated.

**Stack :** Local variables and partial results are store here. Each thread has a private JVM stack created when the thread is created.

**Program register :** Program register holds the address of JVM instruction currently being executed.

**Native method stack :** It contains all native used in application.

**Executive Engine :** Execution engine controls the execute of instructions contained in the methods of the classes.

**Native Method Interface :** Native method interface gives an interface between java code and native code during execution.

**Native Method Libraries :** Native Libraries consist of files required for the execution of native code.

Example:

**public** **class** Hello {

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

System.***out***.println("hello");

}

}

## Steps to Compile and Run your first Java program

**Step 1:** Open a text editor and write the code as above.

**Step 2:** Save the file as Hello.java

**Step 3:** Open command prompt and go to the directory where you saved your first java program assuming it is saved in C drive.

**Step 4:** Type javac Hello.java and press Return**(Enter KEY)** to compile your code. This command will call the Java Compiler asking it to compile the specified file. If there are no errors in the code the command prompt will take you to the next line.

**Step 5:** Now type java Hello on command prompt to run your program.

**Step 6:** You will be able to see **Hello world program** printed on your command prompt.

## Hello World Program using Eclipse

Eclipse is an IDE (Integrated Development Environment) which is used to develop applications. It is design and developed by Eclipse foundation, if you don’t have eclipse download, then download it from its official site by following this download link [Download Eclipse from here](https://www.eclipse.org/downloads/) Here we will see how to create and run **hello world** program using eclipse IDE. It require following steps that consists of **creating project, class file, writing code, running code etc**.

#### **Run Eclipse and Create Project**

Open eclipse startup and then create new project. To create project click on **File** menu and select **Java project** option. It will open a window that ask for project name. Provide the project name and click on the finish button. See the below screenshot.



After creating project, we can see our new created project in the left side bar that looks like below.



### **Create Java Class**

Now create Java class file by **right click** on the **project** and **select class** file option. It will open a window to ask for class name, provide the class name and click on finish button.



### **Write Hello World**

The above created class file includes some line of codes including main method as well. Now we need to write just print statement to print Hello World message.



### **Run The Program**

Now run the program by selecting **Run** menu from the menu bar or use **Ctrl+F11** button combination. After running, it will print Hello World to the console which is just bottom to the program window.



This is a simple program that we run here while using IDE we can create and build large scale of applications. If you are a beginner and not familiar to the Eclipse then don’t worry it is very easy to operate just follow the above steps to create the program.

### **Creating Hello World Example**

Let's create the hello java program:

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

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

Save the above file as Simple.java.

|  |  |
| --- | --- |
| **To compile:** | javac Simple.java |
| **To execute:** | java Simple |

**Output:**

Hello Java

## Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

* **class** keyword is used to declare a class in Java.
* **public** keyword is an access modifier that represents visibility. It means it is visible to all.
* **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
* **void** is the return type of the method. It means it doesn't return any value.
* **main** represents the starting point of the program.
* **String[] args** or **String args[]** is used for [command line argument](https://www.javatpoint.com/command-line-argument). We will discuss it in coming section.
* **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of [System.out.println()](https://www.javatpoint.com/system-out-println-in-java) statement in the coming section.

## What happens at compile time?

At compile time, the 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, the following steps are performed:



**Classloader:** It is the subsystem of JVM that is used to load class files.

Play Videox

**Bytecode Verifier:** Checks the code fragments for illegal code that can violate access rights to objects.

**Interpreter:** Read bytecode stream then execute the instructions.

Example1:

**package** com.Employee;

**public** **class** Student {

**int** rollno=20;

String address="hyd";

**int** age=28;

**void** display() {

System.***out***.println(rollno+" "+address+" "+age);

}

**static** **void** display1() {

System.***out***.println("this is static method");

}

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

Student st=**new** Student();

st.display();

Student.*display1*();

System.***out***.println(st.address);

System.***out***.println(st.age);

}

}

Day2:

Access Modifiers:

There are four types of Java access modifiers:

* 1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
* Example:

|  |
| --- |
| * // Java program to illustrate error while * // using class from different package with * // private modifier * **package** p1; * **class** A * { * **private** **void** display() * { * System.out.println("GeeksforGeeks"); * } * } * **class** B * { * **public** **static** **void** main(String args[]) * { * A obj = **new** A(); * // Trying to access private method * // of another class * obj.display(); * } * } |

**Output:**

error: display() has private access in A

obj.display();

**2.Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Example:

|  |
| --- |
| // Java program to illustrate  // public modifier  **package** p1;  **public** **class** A  {  **public** **void** display()      {          System.out.println("GeeksforGeeks");      }  } |

* Java

|  |
| --- |
| **package** p2;  **import** p1.\*;  **class** B {  **public** **static** **void** main(String args[])      {          A obj = **new** A();          obj.display();      }  } |

**Output:**

GeeksforGeeks

**3.Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package.

Example:

|  |
| --- |
| // Java program to illustrate default modifier  **package** p1;    // Class Geeks is having Default access modifier  **class** Geek  {  **void** display()      {          System.out.println("Hello World!");      }  } |

* Java

|  |
| --- |
| // Java program to illustrate error while  // using class from different package with  // default modifier  **package** p2;  **import** p1.\*;    // This class is having default access modifier  **class** GeekNew  {  **public** **static** **void** main(String args[])      {          // Accessing class Geek from package p1          Geeks obj = **new** Geek();            obj.display();      }  } |

**4.Protected**: The access level of a protected modifier is within the package and outside the package through child class

Example:

|  |
| --- |
| // Java program to illustrate  // protected modifier  **package** p1;    // Class A  **public** **class** A  {  **protected** **void** display()      {          System.out.println("GeeksforGeeks");      }  } |

* Java

|  |
| --- |
| // Java program to illustrate  // protected modifier  **package** p2;  **import** p1.\*; // importing all classes in package p1    // Class B is subclass of A  **class** B **extends** A  {  **public** **static** **void** main(String args[])  {      B obj = **new** B();      obj.display();  }    } |

**Output:**

GeeksforGeeks

DataTypes:

There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).

## 8 Primitive Data Types

### 1. boolean type

* The boolean data type has two possible values, either true or false.
* Default value: false.
* They are usually used for **true/false** conditions.

### Example 1: Java boolean data type

class Main {

public static void main(String[] args) {

boolean flag = true;

System.out.println(flag); // prints true

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

### 2. byte type

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

### Example 2: Java byte data type

class Main {

public static void main(String[] args) {

byte range;

range = 124;

System.out.println(range); // prints 124

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

### 3. short type

* The short data type in Java can have values from **-32768** to **32767** (16-bit signed two's complement integer).
* If it's certain that the value of a variable will be within -32768 and 32767, then it is used instead of other integer data types (int, long).
* Default value: 0

### Example 3: Java short data type

class Main {

public static void main(String[] args) {

short temperature;

temperature = -200;

System.out.println(temperature); // prints -200

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

### 4. int type

* The int data type can have values from **-231** to **231-1** (32-bit signed two's complement integer).
* If you are using Java 8 or later, you can use an unsigned 32-bit integer. This will have a minimum value of 0 and a maximum value of 232-1. To learn more, visit [How to use the unsigned integer in java 8?](http://stackoverflow.com/questions/25556017/how-to-use-the-unsigned-integer-in-java-8)
* Default value: 0

### Example 4: Java int data type

class Main {

public static void main(String[] args) {

int range = -4250000;

System.out.println(range); // print -4250000

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

### 5. long type

* The long data type can have values from **-263** to **263-1** (64-bit signed two's complement integer).
* If you are using Java 8 or later, you can use an unsigned 64-bit integer with a minimum value of **0** and a maximum value of **264-1**.
* Default value: 0

### Example 5: Java long data type

class LongExample {

public static void main(String[] args) {

long range = -42332200000L;

System.out.println(range); // prints -42332200000

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

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

### 6. double type

* 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 6: Java double data type

class Main {

public static void main(String[] args) {

double number = -42.3;

System.out.println(number); // prints -42.3

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

### 7. float type

* The float data type is a single-precision 32-bit floating-point. Learn more about [single-precision and double-precision floating-point](http://stackoverflow.com/questions/801117/whats-the-difference-between-a-single-precision-and-double-precision-floating-p) if you are interested.
* It should never be used for precise values such as currency.
* Default value: 0.0 (0.0f)

### Example 7: Java float data type

class Main {

public static void main(String[] args) {

float number = -42.3f;

System.out.println(number); // prints -42.3

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

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.

If you want to know about single-precision and double-precision,

### 8. char type

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

### Example 8: Java char data type

class Main {

public static void main(String[] args) {

char letter = '\u0051';

System.out.println(letter); // prints Q

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

Here, the Unicode value of Q is **\u0051**. Hence, we get Q as the output.

Here is another example:

class Main {

public static void main(String[] args) {

char letter1 = '9';

System.out.println(letter1); // prints 9

char letter2 = 65;

System.out.println(letter2); // prints A

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

Here, we have assigned 9 as a character (specified by single quotes) to the letter1 variable. However, the letter2 variable is assigned 65 as an integer number (no single quotes).

Hence, A is printed to the output. It is because Java treats characters as an integer and the ASCII value of A is 65.

### String type

Java also provides support for character strings via java.lang.String class. Strings in Java are not primitive types. Instead, they are objects. For example,

String myString = "Java Programming";

Here, myString is an object of the String class.

Methods:

Example:

**package** com.Employee;

**public** **class** Student {

**int** x=10;

**int** y=20;

**public** **void** display()

{

System.***out***.println("sssss");

}

**public** **int** add() {

**int** z=x+y;

**return** z;

}

**public** **int** sum(**int** z,**int** c)

{

**int** s=z+c;

**return** s;

}

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

Student st=**new** Student();

st.display();

**int** m= st.add();

System.***out***.println(m);

**int** d=st.sum(50,70);

System.***out***.println(d);

}

}

In Java, there are two types of methods:

* **User-defined Methods**: We can create our own method based on our requirements.
* **Standard Library Methods**: These are built-in methods in Java that are available to use.

Let's first learn about user-defined methods.

## Declaring a Java Method

The syntax to declare a method is:

returnType methodName() {

// method body

}

Here,

* **returnType** - It specifies what type of value a method returns For example if a method has an int return type then it returns an integer value.  
    
  If the method does not return a value, its return type is void.
* **methodName** - It is an [identifier](https://www.programiz.com/java-programming/keywords-identifiers#identifiers) that is used to refer to the particular method in a program.
* **method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces { }.

For example,

int addNumbers() {

// code

}

In the above example, the name of the method is adddNumbers(). And, the return type is int.

This is the simple syntax of declaring a method. However, the complete syntax of declaring a method is

modifier static returnType nameOfMethod (parameter1, parameter2, ...) {

// method body

}

Here,

* **modifier** - It defines access types whether the method is public, private, and so on. To learn more, visit [Java Access Specifier](https://www.programiz.com/java-programming/access-modifiers).
* **static** - If we use the static keyword, it can be accessed without creating objects.  
    
  For example, the sqrt() method of standard [Math class](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html) is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.
* **parameter1/parameter2** - These are values passed to a method. We can pass any number of arguments to a method.

## Calling a Method in Java

In the above example, we have declared a method named addNumbers(). Now, to use the method, we need to call it.

Here's is how we can call the addNumbers() method.

// calls the method

addNumbers();

Working of Java Method Call

## Example 1: Java Methods

class Main {

// create a method

public int addNumbers(int a, int b) {

int sum = a + b;

// return value

return sum;

}

public static void main(String[] args) {

int num1 = 25;

int num2 = 15;

// create an object of Main

Main obj = new Main();

// calling method

int result = obj.addNumbers(num1, num2);

System.out.println("Sum is: " + result);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Sum is: 40

In the above example, we have created a method named addNumbers(). The method takes two parameters a and b. Notice the line,

int result = obj.addNumbers(num1, num2);

Here, we have called the method by passing two arguments num1 and num2. Since the method is returning some value, we have stored the value in the result variable.

**Note**: The method is not static. Hence, we are calling the method using the object of the class.

## Java Method Return Type

A Java method may or may not return a value to the function call. We use the **return statement** to return any value. For example,

int addNumbers() {

...

return sum;

}

Here, we are returning the variable sum. Since the return type of the function is int. The sum variable should be of int type. Otherwise, it will generate an error.

### Example 2: Method Return Type

class Main {

// create a method

public static int square(int num) {

// return statement

return num \* num;

}

public static void main(String[] args) {

int result;

// call the method

// store returned value to result

result = square(10);

System.out.println("Squared value of 10 is: " + result);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Squared value of 10 is: 100

In the above program, we have created a method named square(). The method takes a number as its parameter and returns the square of the number.

Here, we have mentioned the return type of the method as int. Hence, the method should always return an integer value.

Representation of the Java method returning a value

**Note**: If the method does not return any value, we use the void keyword as the return type of the method. For example,

public void square(int a) {

int square = a \* a;

System.out.println("Square is: " + square);

}

## Method Parameters in Java

A method parameter is a value accepted by the method. As mentioned earlier, a method can also have any number of parameters. For example,

// method with two parameters

int addNumbers(int a, int b) {

// code

}

// method with no parameter

int addNumbers(){

// code

}

If a method is created with parameters, we need to pass the corresponding values while calling the method. For example,

// calling the method with two parameters

addNumbers(25, 15);

// calling the method with no parameters

addNumbers()

### Example 3: Method Parameters

class Main {

// method with no parameter

public void display1() {

System.out.println("Method without parameter");

}

// method with single parameter

public void display2(int a) {

System.out.println("Method with a single parameter: " + a);

}

public static void main(String[] args) {

// create an object of Main

Main obj = new Main();

// calling method with no parameter

obj.display1();

// calling method with the single parameter

obj.display2(24);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Method without parameter

Method with a single parameter: 24

Here, the parameter of the method is int. Hence, if we pass any other data type instead of int, the compiler will throw an error. It is because Java is a strongly typed language.

**Note**: The argument 24 passed to the display2() method during the method call is called the actual argument.

The parameter num accepted by the method definition is known as a formal argument. We need to specify the type of formal arguments. And, the type of actual arguments and formal arguments should always match.

## Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (\*.jar) file with JVM and JRE.

For example,

* print() is a method of java.io.PrintSteam. The print("...") method prints the string inside quotation marks.
* sqrt() is a method of Math class. It returns the square root of a number.

Here's a working example:

### Example 4: Java Standard Library Method

public class Main {

public static void main(String[] args) {

// using the sqrt() method

System.out.print("Square root of 4 is: " + Math.sqrt(4));

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Square root of 4 is: 2.0

To learn more about standard library methods, visit [Java Library Methods](https://www.programiz.com/java-programming/library).

## What are the advantages of using methods?

**1.** The main advantage is **code reusability**. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times".

### Example 5: Java Method for Code Reusability

public class Main {

// method defined

private static int getSquare(int x){

return x \* x;

}

public static void main(String[] args) {

for (int i = 1; i <= 5; i++) {

// method call

int result = getSquare(i);

System.out.println("Square of " + i + " is: " + result);

}

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Square of 1 is: 1

Square of 2 is: 4

Square of 3 is: 9

Square of 4 is: 16

Square of 5 is: 25

In the above program, we have created the method named getSquare() to calculate the square of a number. Here, the method is used to calculate the square of numbers less than **6**.

Hence, the same method is used again and again.

**2.** Methods make code more **readable and easier** to debug. Here, the getSquare() method keeps the code to compute the square in a block. Hence, makes it more readable.

## What is a Constructor?

A constructor in Java is similar to a method that is invoked when an object of the class is created.

 a constructor has the same name as that of the class and does not have any return type. For example,

class Test {

Test() {

// constructor body

}

}

Here, Test() is a constructor. It has the same name as that of the class and doesn't have a return type.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

### **Rules for creating Java constructor**

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

### Types of Constructor

In Java, constructors can be divided into 3 types:

1. No-Arg Constructor
2. Parameterized Constructor
3. Default Constructor

## 1. Java No-Arg Constructors

Similar to methods, a Java constructor may or may not have any parameters (arguments).

If a constructor does not accept any parameters, it is known as a no-argument constructor. For example,

private Constructor() {

// body of the constructor

}

### Example 2: Java private no-arg constructor

class Main {

int i;

// constructor with no parameter

private Main() {

i = 5;

System.out.println("Constructor is called");

}

public static void main(String[] args) {

// calling the constructor without any parameter

Main obj = new Main();

System.out.println("Value of i: " + obj.i);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Constructor is called

Value of i: 5

In the above example, we have created a constructor Main(). Here, the constructor does not accept any parameters. Hence, it is known as a no-arg constructor.

**Notice that we have declared the constructor as private.**

Once a constructor is declared private, it cannot be accessed from outside the class. So, creating objects from outside the class is prohibited using the private constructor.

Here, we are creating the object inside the same class. Hence, the program is able to access the constructor. To learn more, visit [Java Implement Private Constructor](https://www.programiz.com/java-programming/examples/private-constructor-implementation).

However, if we want to create objects outside the class, then we need to declare the constructor as public.

### Example 3: Java public no-arg constructors

class Company {

String name;

// public constructor

public Company() {

name = "Programiz";

}

}

class Main {

public static void main(String[] args) {

// object is created in another class

Company obj = new Company();

System.out.println("Company name = " + obj.name);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Company name = Programiz

**Recommended Reading:** [Java Access Modifier](https://www.programiz.com/java-programming/access-modifiers)

## 2. Java Parameterized Constructor

A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors (constructor with parameters).

### Example 4: Parameterized constructor

class Main {

String languages;

// constructor accepting single value

Main(String lang) {

languages = lang;

System.out.println(languages + " Programming Language");

}

public static void main(String[] args) {

// call constructor by passing a single value

Main obj1 = new Main("Java");

Main obj2 = new Main("Python");

Main obj3 = new Main("C");

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Java Programming Language

Python Programming Language

C Programming Language

In the above example, we have created a constructor named Main(). Here, the constructor takes a single parameter. Notice the expression,

Main obj1 = new Main("Java");

Here, we are passing the single value to the constructor. Based on the argument passed, the language variable is initialized inside the constructor.

## 3. Java Default Constructor

If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor.



### Example 5: Default Constructor

class Main {

int a;

boolean b;

public static void main(String[] args) {

// A default constructor is called

Main obj = new Main();

System.out.println("Default Value:");

System.out.println("a = " + obj.a);

System.out.println("b = " + obj.b);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Default Value:

a = 0

b = false

Here, we haven't created any constructors. Hence, the Java compiler automatically creates the default constructor.

The default constructor initializes any uninitialized instance variables with default values.

|  |  |
| --- | --- |
| Type | Default Value |
| boolean | false |
| Byte | 0 |
| Short | 0 |
| Int | 0 |
| Long | 0L |
| Char | \u0000 |
| Float | 0.0f |
| double | 0.0d |
| object | Reference null |

In the above program, the variables a and b are initialized with default value **0** and false respectively.

The above program is equivalent to:

class Main {

int a;

boolean b;

Main() {

a = 0;

b = false;

}

public static void main(String[] args) {

// call the constructor

Main obj = new Main();

System.out.println("Default Value:");

System.out.println("a = " + obj.a);

System.out.println("b = " + obj.b);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**The output of the program is the same as Example 5.**

## Important Notes on Java Constructors

* Constructors are invoked implicitly when you instantiate objects.
* The two rules for creating a constructor are:  
  The name of the constructor should be the same as the class.  
  A Java constructor must not have a return type.
* If a class doesn't have a constructor, the Java compiler automatically creates a **default constructor** during run-time. The default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0
* Constructor types:  
  **No-Arg Constructor** - a constructor that does not accept any arguments  
  **Parameterized constructor** - a constructor that accepts arguments  
  **Default Constructor** - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
* A constructor cannot be abstract or static or final.
* A constructor can be overloaded but can not be overridden.

## Constructors Overloading in Java

Similar to [Java method overloading](https://www.programiz.com/java-programming/method-overloading), we can also create two or more constructors with different parameters. This is called constructors overloading.

### Example 6: Java Constructor Overloading

class Main {

String language;

// constructor with no parameter

Main() {

this.language = "Java";

}

// constructor with a single parameter

Main(String language) {

this.language = language;

}

public void getName() {

System.out.println("Programming Langauage: " + this.language);

}

public static void main(String[] args) {

// call constructor with no parameter

Main obj1 = new Main();

// call constructor with a single parameter

Main obj2 = new Main("Python");

obj1.getName();

obj2.getName();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Programming Language: Java

Programming Language: Python

In the above example, we have two constructors: Main() and Main(String language). Here, both the constructor initialize the value of the variable language with different values.

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

TypeCasting:

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically.

Type casting is when you assign a value of one primitive data type to another type.

Order:

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

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char -> short -> byte

**WideningTypeCastingExample.java**

1. **public** **class** WideningTypeCastingExample
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **int** x = 7;
6. //automatically converts the integer type into long type
7. **long** y = x;
8. //automatically converts the long type into float type
9. **float** z = y;
10. System.out.println("Before conversion, int value "+x);
11. System.out.println("After conversion, long value "+y);
12. System.out.println("After conversion, float value "+z);
13. }
14. }

**Output**

Before conversion, the value is: 7

After conversion, the long value is: 7

After conversion, the float value is: 7.0

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

### **Narrowing Type Casting**

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

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

Let's see an example of narrowing type casting.

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

**NarrowingTypeCastingExample.java**

1. **public** **class** NarrowingTypeCastingExample
2. {
3. **public** **static** **void** main(String args[])
4. {
5. **double** d = 166.66;
6. //converting double data type into long data type
7. **long** l = (**long**)d;
8. //converting long data type into int data type
9. **int** i = (**int**)l;
10. System.out.println("Before conversion: "+d);
11. //fractional part lost
12. System.out.println("After conversion into long type: "+l);
13. //fractional part lost
14. System.out.println("After conversion into int type: "+i);
15. }
16. }

**Output**

Before conversion: 166.66

After conversion into long type: 166

After conversion into int type: 166

### **Types of Variables**

There are three types of variables in [Java](https://www.javatpoint.com/java-tutorial):

* local variable
* instance variable
* static variable



#### **1) Local Variable**

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

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

#### **2) Instance Variable**

A variable declared inside the class but outside the body of the method, is called an

#### **3) Static variable**

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

### **Example to understand the types of variables in java**

1. **public** **class** A
2. {
3. **static** **int** m=100;//static variable
4. **void** method()
5. {
6. **int** n=90;//local variable
7. }
8. **public** **static** **void** main(String args[])
9. {
10. **int** data=50;//instance variable
11. }
12. }//end of class

### **Java Variable Example: Add Two Numbers**

1. **public** **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. **int** b=10;
5. **int** c=a+b;
6. System.out.println(c);
7. }
8. }

**Output:**

20

### **Java Variable Example: Widening**

1. **public** **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **int** a=10;
4. **float** f=a;
5. System.out.println(a);
6. System.out.println(f);
7. }}

**Output:**

10

10.0

### **Java Variable Example: Narrowing (Typecasting)**

1. **public** **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **float** f=10.5f;
4. //int a=f;//Compile time error
5. **int** a=(**int**)f;
6. System.out.println(f);
7. System.out.println(a);
8. }}

**Output:**

10.5

10

### **Java Variable Example: Overflow**

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. //Overflow
4. **int** a=130;
5. **byte** b=(**byte**)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}

**Output:**

130

-126

### **Java Variable Example: Adding Lower Type**

1. **class** Simple{
2. **public** **static** **void** main(String[] args){
3. **byte** a=10;
4. **byte** b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. **byte** c=(**byte**)(a+b);
7. System.out.println(c);
8. }}

**Output:**

1. //Java Program to demonstrate the use of static variable
2. **class** Student{
3. **int** rollno;//instance variable
4. String name;
5. **static** String college ="ITS";//static variable
6. //constructor
7. Student(**int** r, String n){
8. rollno = r;
9. name = n;
10. }
11. //method to display the values
12. **void** display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. **public** **class** TestStaticVariable1{
16. **public** **static** **void** main(String args[]){
17. Student s1 = **new** Student(111,"Karan");
18. Student s2 = **new** Student(222,"Aryan");
19. //we can change the college of all objects by the single line of code
20. //Student.college="BBDIT";
21. s1.display();
22. s2.display();
23. }
24. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestStaticVariable1)

Output:

111 Karan ITS

222 Aryan ITS



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.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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.

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }

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

Student.java

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y > 20) {
6. System.out.println("x + y is greater than 20");
7. }
8. }
9. }

**Output:**

x + y is greater than 20

### **2) if-else statement**

The [if-else statement](https://www.javatpoint.com/java-if-else) 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:**

1. **if**(condition) {
2. statement 1; //executes when condition is true
3. }
4. **else**{
5. statement 2; //executes when condition is false
6. }

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. **int** x = 10;
4. **int** y = 12;
5. **if**(x+y < 10) {
6. System.out.println("x + y is less than      10");
7. }   **else** {
8. System.out.println("x + y is greater than 20");
9. }
10. }
11. }

**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.

1. **if**(condition 1) {
2. statement 1; //executes when condition 1 is true
3. }
4. **else** **if**(condition 2) {
5. statement 2; //executes when condition 2 is true
6. }
7. **else** {
8. statement 2; //executes when all the conditions are false
9. }

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. String city = "Delhi";
4. **if**(city == "Meerut") {
5. System.out.println("city is meerut");
6. }**else** **if** (city == "Noida") {
7. System.out.println("city is noida");
8. }**else** **if**(city == "Agra") {
9. System.out.println("city is agra");
10. }**else** {
11. System.out.println(city);
12. }
13. }
14. }

**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.

1. **if**(condition 1) {
2. statement 1; //executes when condition 1 is true
3. **if**(condition 2) {
4. statement 2; //executes when condition 2 is true
5. }
6. **else**{
7. statement 2; //executes when condition 2 is false
8. }
9. }

Consider the following example.

**Student.java**

1. **public** **class** Student {
2. **public** **static** **void** main(String[] args) {
3. String address = "Delhi, India";
5. **if**(address.endsWith("India")) {
6. **if**(address.contains("Meerut")) {
7. System.out.println("Your city is Meerut");
8. }**else** **if**(address.contains("Noida")) {
9. System.out.println("Your city is Noida");
10. }**else** {
11. System.out.println(address.split(",")[0]);
12. }
13. }**else** {
14. System.out.println("You are not living in India");
15. }
16. }
17. }

**Output:**

Delhi

### **Switch Statement:**

In Java, [Switch statements](https://www.javatpoint.com/java-switch) 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.

1. **switch** (expression){
2. **case** value1:
3. statement1;
4. **break**;
5. .
6. .
7. .
8. **case** valueN:
9. statementN;
10. **break**;
11. **default**:
12. **default** statement;
13. }

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

**Student.java**

1. **public** **class** Student **implements** Cloneable {
2. **public** **static** **void** main(String[] args) {
3. **int** num = 2;
4. **switch** (num){
5. **case** 0:
6. System.out.println("number is 0");
7. **break**;
8. **case** 1:
9. System.out.println("number is 1");
10. **break**;
11. **default**:
12. System.out.println(num);
13. }
14. }
15. }

**Output:**

2

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 switch permits only int, string, and Enum type variables to be used.

### **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](https://www.javatpoint.com/java-for-loop) is similar to [C](https://www.javatpoint.com/c-programming-language-tutorial) and [C++](https://www.javatpoint.com/cpp-tutorial). 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.

1. **for**(initialization, condition, increment/decrement) {
2. //block of statements
3. }

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**

1. **public** **class** Calculattion {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** sum = 0;
5. **for**(**int** j = 1; j<=10; j++) {
6. sum = sum + j;
7. }
8. System.out.println("The sum of first 10 natural numbers is " + sum);
9. }
10. }

**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.

1. **for**(data\_type var : array\_name/collection\_name){
2. //statements
3. }

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

**Calculation.java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. String[] names = {"Java","C","C++","Python","JavaScript"};
5. System.out.println("Printing the content of the array names:\n");
6. **for**(String name:names) {
7. System.out.println(name);
8. }
9. }
10. }

**Output:**

Printing the content of the array names:

Java

C

C++

Python

JavaScript

### **Java while loop**

The [while loop](https://www.javatpoint.com/java-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.

1. **while**(condition){
2. //looping statements
3. }

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



Consider the following example.

**Calculation .java**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **while**(i<=10) {
7. System.out.println(i);
8. i = i + 2;
9. }
10. }
11. }

**Output:**

Printing the list of first 10 even numbers

0

2

4

6

8

10

### **Java do-while loop**

The [do-while loop](https://www.javatpoint.com/java-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.

1. **do**
2. {
3. //statements
4. } **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**

1. **public** **class** Calculation {
2. **public** **static** **void** main(String[] args) {
3. // TODO Auto-generated method stub
4. **int** i = 0;
5. System.out.println("Printing the list of first 10 even numbers \n");
6. **do** {
7. System.out.println(i);
8. i = i + 2;
9. }**while**(i<=10);
10. }
11. }

**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](https://www.javatpoint.com/java-break) 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**

1. **public** **class** BreakExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
5. **for**(**int** i = 0; i<= 10; i++) {
6. System.out.println(i);
7. **if**(i==6) {
8. **break**;
9. }
10. }
11. }
12. }

**Output:**

0

1

2

3

4

5

6

**break statement example with labeled for loop**

**Calculation.java**

1. **public** **class** Calculation {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
5. a:
6. **for**(**int** i = 0; i<= 10; i++) {
7. b:
8. **for**(**int** j = 0; j<=15;j++) {
9. c:
10. **for** (**int** k = 0; k<=20; k++) {
11. System.out.println(k);
12. **if**(k==5) {
13. **break** a;
14. }
15. }
16. }
18. }
19. }

22. }

**Output:**

0

1

2

3

4

5

### **Java continue statement**

Unlike break statement, the [continue statement](https://www.javatpoint.com/java-continue) 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.

1. **public** **class** ContinueExample {
3. **public** **static** **void** main(String[] args) {
4. // TODO Auto-generated method stub
6. **for**(**int** i = 0; i<= 2; i++) {
8. **for** (**int** j = i; j<=5; j++) {
10. **if**(j == 4) {
11. **continue**;
12. }
13. System.out.println(j);
14. }
15. }
16. }
18. }

**Output:**

0

1

2

3

5

1

2

3

5

2

3

5

# **Java Naming Convention:**

Naming Conventions of the Different Identifiers

The following table shows the popular conventions used for the different identifiers.

|  |  |  |
| --- | --- | --- |
| **Identifiers Type** | **Naming Rules** | **Examples** |
| Class | It should start with the uppercase letter. It should be a noun such as Color, Button, System, Thread, etc. Use appropriate words, instead of acronyms. | public class **Employee** { //code snippet } |
| Interface | It should start with the uppercase letter. It should be an adjective such as Runnable, Remote, ActionListener. Use appropriate words, instead of acronyms. | interface **Printable** { //code snippet } |
| Method | It should start with lowercase letter. It should be a verb such as main(), print(), println(). If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed(). | class Employee { // method void **draw()** { //code snippet } } |
| Variable | It should start with a lowercase letter such as id, name. It should not start with the special characters like & (ampersand), $ (dollar), \_ (underscore). If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName. Avoid using one-character variables such as x, y, z. | class Employee { // variable int **id**; //code snippet } |
| Package | It should be a lowercase letter such as java, lang. If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang. | //package package **com.javapoint;** class Employee { //code snippet } |
| Constant | It should be in uppercase letters such as RED, YELLOW. If the name contains multiple words, it should be separated by an underscore(\_) such as MAX\_PRIORITY. It may contain digits but not as the first letter. | class Employee { //constant static final int **MIN\_AGE** = 18; //code snippet } |

OOPS:

1. Pillars of OOPs
   * + - Class
       - Object
   * [Abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2/)
   * [Encapsulation](https://www.geeksforgeeks.org/encapsulation-in-java/)
   * [Inheritance](https://www.geeksforgeeks.org/inheritance-in-java/)
   * [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-java/)
     + Compile-time polymorphism
     + Runtime polymorphism

Inheritance:

Inheritance is one of the key features of OOP that allows us to create a new class from an existing class.

# **Inheritance in Java**

1. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
2. [Types of Inheritance](https://www.javatpoint.com/inheritance-in-java#inheritancetypes)
3. [Why multiple inheritance is not possible in Java in case of class?](https://www.javatpoint.com/inheritance-in-java#inheritancenotmultiple)

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.

### **Why use inheritance in java**

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### **Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### **The syntax of Java Inheritance**

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

x

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

### **Java Inheritance Example**



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

**package** com.sample;

**public** **class** Employee {

**float** salary = 40000;

}

**class** Programmer **extends** Employee {

**int** bonus = 10000;

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

Programmer p = **new** Programmer();

System.***out***.println("Programmer salary is:" + p.salary);

System.***out***.println("Bonus of Programmer is:" + p.bonus);

}

}

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

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



#### **Note: Multiple inheritance is not supported in Java through class.**

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



## Single Inheritance Example

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

*File: TestInheritance.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

Output:

barking...

eating...

## Multilevel Inheritance Example

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

*File: TestInheritance2.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

Output:

weeping...

barking...

eating...

## Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

*File: TestInheritance3.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

Output:

meowing...

eating...

## Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

1. **class** A{
2. **void** msg(){System.out.println("Hello");}
3. }
4. **class** B{
5. **void** msg(){System.out.println("Welcome");}
6. }
7. **class** C **extends** A,B{//suppose if it were
9. **public** **static** **void** main(String args[]){
10. C obj=**new** C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

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

Compile Time Error

Abstraction:

Abstraction is a feature of OOPs. The feature allows us to hide the implementation detail from the user and shows only the functionality of the programming to the user.

Let's understand the abstraction with the help of a real-world example. The best example of abstraction is a car. When we derive a car, we do not know how is the car moving or how internal components are working? But we know how to derive a car. It means it is not necessary to know how the car is working, but it is important how to derive a car. The same is an abstraction.

We can achieve the abstraction in two ways:

1)Using Abstract Class

2)Using Interface

Abstract:

### **Using Abstract Class**

# **Abstract class in Java**

A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

### **Abstraction in Java**

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Abstraction lets you focus on what the [object](https://www.javatpoint.com/object-and-class-in-java) does instead of how it does it.

### **Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

### **Abstract class in Java**

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### **Points to Remember**

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.



**Example of abstract class**

1. **abstract** **class** A{}

### **Abstract Method in Java**

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

1. **abstract** **void** printStatus();//no method body and abstract

### **Example of Abstract class that has an abstract method**

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

1. **abstract** **class** Bike{
2. **abstract** **void** run();
3. }
4. **class** Honda4 **extends** Bike{
5. **void** run(){System.out.println("running safely");}
6. **public** **static** **void** main(String args[]){
7. Honda4 obj = **new** Honda4();
8. obj.run();
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Honda4)

running safely

### **Understanding the real scenario of Abstract class**

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

*File: TestAbstraction1.java*

1. **abstract** **class** Shape{
2. **abstract** **void** draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle1 **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. **class** TestAbstraction1{
13. **public** **static** **void** main(String args[]){
14. Circle1 s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
15. s.draw();
16. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)

drawing circle

### **Another example of Abstract class in java**

*File: TestBank.java*

1. **abstract** **class** Bank{
2. **abstract** **int** getRateOfInterest();
3. }
4. **class** SBI **extends** Bank{
5. **int** getRateOfInterest(){**return** 7;}
6. }
7. **class** PNB **extends** Bank{
8. **int** getRateOfInterest(){**return** 8;}
9. }
11. **class** TestBank{
12. **public** **static** **void** main(String args[]){
13. Bank b;
14. b=**new** SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=**new** PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}

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

Rate of Interest is: 7 %

Rate of Interest is: 8 %

### **Abstract class having constructor, data member and methods**

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

*File: TestAbstraction2.java*

1. //Example of an abstract class that has abstract and non-abstract methods
2. **abstract** **class** Bike{
3. Bike(){System.out.println("bike is created");}
4. **abstract** **void** run();
5. **void** changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a Child class which inherits Abstract class
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. **class** TestAbstraction2{
13. **public** **static** **void** main(String args[]){
14. Honda obj = **new** Honda();
15. obj.run();
16. obj.changeGear();
17. }
18. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction2)

bike is created

running safely..

gear changed

#### **Rule: If there is an abstract method in a class, that class must be abstract.**

1. **class** Bike12{
2. **abstract** **void** run();
3. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Bike12)

compile time error

#### **Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.**

Interfaces:

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the interface keyword to create an interface in Java. For example,

interface Language {

public void getType();

public void getVersion();

}

Here,

* Language is an interface.
* It includes abstract methods: getType() and getVersion().

## Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the implements keyword to implement an interface.

### Example 1: Java Interface

interface Polygon {

void getArea(int length, int breadth);

}

// implement the Polygon interface

class Rectangle implements Polygon {

// implementation of abstract method

public void getArea(int length, int breadth) {

System.out.println("The area of the rectangle is " + (length \* breadth));

}

}

class Main {

public static void main(String[] args) {

Rectangle r1 = new Rectangle();

r1.getArea(5, 6);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

The area of the rectangle is 30

In the above example, we have created an interface named Polygon. The interface contains an abstract method getArea().

Here, the Rectangle class implements Polygon. And, provides the implementation of the getArea() method.

### Example 2: Java Interface

// create an interface

interface Language {

void getName(String name);

}

// class implements interface

class ProgrammingLanguage implements Language {

// implementation of abstract method

public void getName(String name) {

System.out.println("Programming Language: " + name);

}

}

class Main {

public static void main(String[] args) {

ProgrammingLanguage language = new ProgrammingLanguage();

language.getName("Java");

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Programming Language: Java

In the above example, we have created an interface named Language. The interface includes an abstract method getName().

Here, the ProgrammingLanguage class implements the interface and provides the implementation for the method.

### Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

interface A {

// members of A

}

interface B {

// members of B

}

class C implements A, B {

// abstract members of A

// abstract members of B

}

## Extending an Interface

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

interface Line {

// members of Line interface

}

// extending interface

interface Polygon extends Line {

// members of Polygon interface

// members of Line interface

}

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

### Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

interface A {

...

}

interface B {

...

}

interface C extends A, B {

...

}

## Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

* Similar to abstract classes, interfaces help us to achieve **abstraction in Java**.  
    
  Here, we know getArea() calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of getArea() is independent of one another.
* Interfaces **provide specifications** that a class (which implements it) must follow.  
    
  In our previous example, we have used getArea() as a specification inside the interface Polygon. This is like setting a rule that we should be able to get the area of every polygon.  
    
  Now any class that implements the Polygon interface must provide an implementation for the getArea() method.
* Interfaces are also used to achieve multiple inheritance in Java. For example,
* interface Line {
* …
* }
* interface Polygon {
* …
* }
* class Rectangle implements Line, Polygon {
* …

}

Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

**Note**: All the methods inside an interface are implicitly public and all fields are implicitly public static final. For example,

interface Language {

// by default public static final

String type = "programming language";

// by default public

void getName();

}

## default methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the default keyword. For example,

public default void getSides() {

// body of getSides()

}

### Why default methods?

Let's take a scenario to understand why default methods are introduced in Java.

Suppose, we need to add a new method in an interface.

We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.

If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well.

To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

Let's take an example to have a better understanding of default methods.

### Example: Default Method in Java Interface

interface Polygon {

void getArea();

// default method

default void getSides() {

System.out.println("I can get sides of a polygon.");

}

}

// implements the interface

class Rectangle implements Polygon {

public void getArea() {

int length = 6;

int breadth = 5;

int area = length \* breadth;

System.out.println("The area of the rectangle is " + area);

}

// overrides the getSides()

public void getSides() {

System.out.println("I have 4 sides.");

}

}

// implements the interface

class Square implements Polygon {

public void getArea() {

int length = 5;

int area = length \* length;

System.out.println("The area of the square is " + area);

}

}

class Main {

public static void main(String[] args) {

// create an object of Rectangle

Rectangle r1 = new Rectangle();

r1.getArea();

r1.getSides();

// create an object of Square

Square s1 = new Square();

s1.getArea();

s1.getSides();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

The area of the rectangle is 30

I have 4 sides.

The area of the square is 25

I can get sides of a polygon.

In the above example, we have created an interface named Polygon. It has a default method getSides() and an abstract method getArea().

Here, we have created two classes Rectangle and Square that implement Polygon.

The Rectangle class provides the implementation of the getArea() method and overrides the getSides() method. However, the Square class only provides the implementation of the getArea() method.

Now, while calling the getSides() method using the Rectangle object, the overridden method is called. However, in the case of the Square object, the default method is called.

## private and static Methods in Interface

The Java 8 also added another feature to include static methods inside an interface.

Similar to a class, we can access static methods of an interface using its references. For example,

// create an interface

interface Polygon {

staticMethod(){..}

}

// access static method

Polygon.staticMethod();

**Note**: With the release of Java 9, private methods are also supported in interfaces.

We cannot create objects of an interface. Hence, private methods are used as helper methods that provide support to other methods in interfaces.

### Practical Example of Interface

Let's see a more practical example of Java Interface.

// To use the sqrt function

import java.lang.Math;

interface Polygon {

void getArea();

// calculate the perimeter of a Polygon

default void getPerimeter(int... sides) {

int perimeter = 0;

for (int side: sides) {

perimeter += side;

}

System.out.println("Perimeter: " + perimeter);

}

}

class Triangle implements Polygon {

private int a, b, c;

private double s, area;

// initializing sides of a triangle

Triangle(int a, int b, int c) {

this.a = a;

this.b = b;

this.c = c;

s = 0;

}

// calculate the area of a triangle

public void getArea() {

s = (double) (a + b + c)/2;

area = Math.sqrt(s\*(s-a)\*(s-b)\*(s-c));

System.out.println("Area: " + area);

}

}

class Main {

public static void main(String[] args) {

Triangle t1 = new Triangle(2, 3, 4);

// calls the method of the Triangle class

t1.getArea();

// calls the method of Polygon

t1.getPerimeter(2, 3, 4);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Area: 2.9047375096555625

Perimeter: 9

In the above program, we have created an interface named Polygon. It includes a default method getPerimeter() and an abstract method getArea().

We can calculate the perimeter of all polygons in the same manner so we implemented the body of getPerimeter() in Polygon.

Now, all polygons that implement Polygon can use getPerimeter() to calculate perimeter.

However, the rule for calculating the area is different for different polygons. Hence, getArea() is included without implementation.

Any class that implements Polygon must provide an implementation of getArea().

# **Polymorphism in Java:**

Polymorphism is an important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

## Example: Java Polymorphism

class Polygon {

// method to render a shape

public void render() {

System.out.println("Rendering Polygon...");

}

}

class Square extends Polygon {

// renders Square

public void render() {

System.out.println("Rendering Square...");

}

}

class Circle extends Polygon {

// renders circle

public void render() {

System.out.println("Rendering Circle...");

}

}

class Main {

public static void main(String[] args) {

// create an object of Square

Square s1 = new Square();

s1.render();

// create an object of Circle

Circle c1 = new Circle();

c1.render();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**

Rendering Square...

Rendering Circle...

In the above example, we have created a superclass: Polygon and two subclasses: Square and Circle. Notice the use of the render() method.

The main purpose of the render() method is to render the shape. However, the process of rendering a square is different than the process of rendering a circle.

Hence, the render() method behaves differently in different classes. Or, we can say render() is polymorphic.

### Why Polymorphism?

Polymorphism allows us to create consistent code. In the previous example, we can also create different methods: renderSquare() and renderCircle() to render Square and Circle, respectively.

This will work perfectly. However, for every shape, we need to create different methods. It will make our code inconsistent.

To solve this, polymorphism in Java allows us to create a single method render() that will behave differently for different shapes.

**Note**: The print() method is also an example of polymorphism. It is used to print values of different types like char, int, string, etc.

We can achieve polymorphism in Java using the following ways:

1. [Method Overriding](https://www.programiz.com/java-programming/method-overriding)
2. [Method Overloading](https://www.programiz.com/java-programming/method-overloading)
3. Operator Overloading

## Java Method Overriding

During [inheritance in Java](https://www.programiz.com/java-programming/inheritance), if the same method is present in both the superclass and the subclass. Then, the method in the subclass overrides the same method in the superclass. This is called method overriding.

In this case, the same method will perform one operation in the superclass and another operation in the subclass. For example,

### Example 1: Polymorphism using method overriding

class Language {

public void displayInfo() {

System.out.println("Common English Language");

}

}

class Java extends Language {

@Override

public void displayInfo() {

System.out.println("Java Programming Language");

}

}

class Main {

public static void main(String[] args) {

// create an object of Java class

Java j1 = new Java();

j1.displayInfo();

// create an object of Language class

Language l1 = new Language();

l1.displayInfo();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

Java Programming Language

Common English Language

In the above example, we have created a superclass named Language and a subclass named Java. Here, the method displayInfo() is present in both Language and Java.

The use of displayInfo() is to print the information. However, it is printing different information in Language and Java.

Based on the object used to call the method, the corresponding information is printed.

Working of Java Polymorphism

**Note**: The method that is called is determined during the execution of the program. Hence, method overriding is a **run-time polymorphism**.

## 2. Java Method Overloading

In a Java class, we can create methods with the same name if they differ in parameters. For example,

void func() { ... }

void func(int a) { ... }

float func(double a) { ... }

float func(int a, float b) { ... }

This is known as method overloading in Java. Here, the same method will perform different operations based on the parameter.

### Example 3: Polymorphism using method overloading

class Pattern {

// method without parameter

public void display() {

for (int i = 0; i < 10; i++) {

System.out.print("\*");

}

}

// method with single parameter

public void display(char symbol) {

for (int i = 0; i < 10; i++) {

System.out.print(symbol);

}

}

}

class Main {

public static void main(String[] args) {

Pattern d1 = new Pattern();

// call method without any argument

d1.display();

System.out.println("\n");

// call method with a single argument

d1.display('#');

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

\*\*\*\*\*\*\*\*\*\*

##########

In the above example, we have created a class named Pattern. The class contains a method named display() that is overloaded.

// method with no arguments

display() {...}

// method with a single char type argument

display(char symbol) {...}

Here, the main function of display() is to print the pattern. However, based on the arguments passed, the method is performing different operations:

* prints a pattern of \*, if no argument is passed or
* prints pattern of the parameter, if a single char type argument is passed.

**Note**: The method that is called is determined by the compiler. Hence, it is also known as compile-time polymorphism.

## 3. Java Operator Overloading

Some operators in Java behave differently with different operands. For example,

* + operator is overloaded to perform numeric addition as well as string concatenation, and
* operators like &, |, and ! are overloaded for logical and bitwise operations.

Let's see how we can achieve polymorphism using operator overloading.

The + operator is used to add two entities. However, in Java, the + operator performs two operations.

1. When + is used with numbers (integers and floating-point numbers), it performs mathematical addition. For example,

int a = 5;

int b = 6;

// + with numbers

int sum = a + b; // Output = 11

2. When we use the + operator with strings, it will perform string concatenation (join two strings). For example,

String first = "Java ";

String second = "Programming";

// + with strings

name = first + second; // Output = Java Programming

Here, we can see that the + operator is overloaded in Java to perform two operations: **addition** and **concatenation**.

**Note**: In languages like C++, we can define operators to work differently for different operands. However, Java doesn't support user-defined operator overloading.

## Polymorphic Variables:

## A reference variable of the super class can refer to a sub class object

A variable is called polymorphic if it refers to different values under different conditions.

Object variables (instance variables) represent the behavior of polymorphic variables in Java. It is because object variables of a class can refer to objects of its class as well as objects of its subclasses.

### Example: Polymorphic Variables

class ProgrammingLanguage {

public void display() {

System.out.println("I am Programming Language.");

}

}

class Java extends ProgrammingLanguage {

@Override

public void display() {

System.out.println("I am Object-Oriented Programming Language.");

}

}

class Main {

public static void main(String[] args) {

// declare an object variable

ProgrammingLanguage pl;

// create object of ProgrammingLanguage

pl = new ProgrammingLanguage();

pl.display();

// create object of Java class

pl = new Java();

pl.display();

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**Output**:

I am Programming Language.

I am Object-Oriented Programming Language.

In the above example, we have created an object variable pl of the ProgrammingLanguage class. Here, pl is a polymorphic variable. This is because,

* In statement pl = new ProgrammingLanguage(), pl refer to the object of the ProgrammingLanguage class.
* And, in statement pl = new Java(), pl refer to the object of the Java class.

We have one parent class, ‘Account’ with function of deposit and withdraw. Account has 2 child classes

The operation of deposit and withdraw is same for Saving and Checking accounts. So the inherited methods from Account class will work.



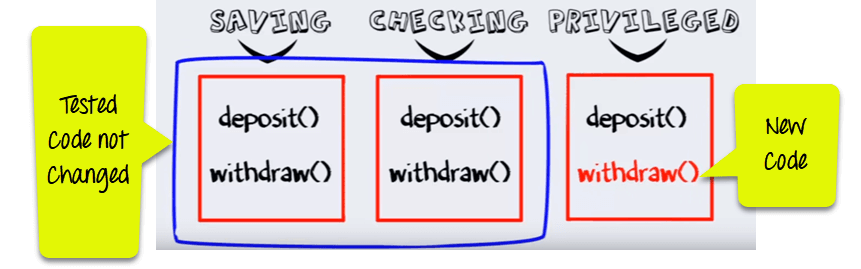
Java Polymorphism Example

### Change in Software Requirement

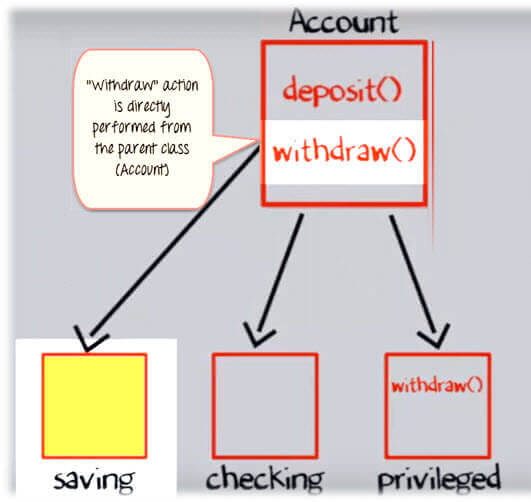
There is a change in the requirement specification, something that is so common in the software industry. You are supposed to add functionality privileged Banking Account with Overdraft Facility.

For a background, overdraft is a facility where you can withdraw an amount more than available the balance in your account.

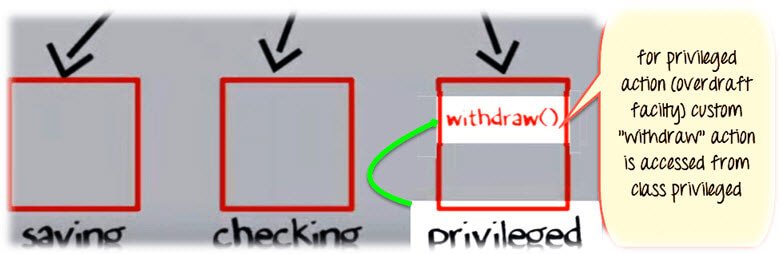
So, withdraw method for privileged needs to implemented afresh. But you do not change the tested piece of code in Savings and Checking account. This is advantage of OOPS



**Step 1)** Such that when the “withdrawn” method for saving account is called a method from parent account class is executed



**Step 2)**But when the “Withdraw” method for the privileged account (overdraft facility) is called withdraw method defined in the privileged class is executed. This is **Polymorphism in OOPs.**

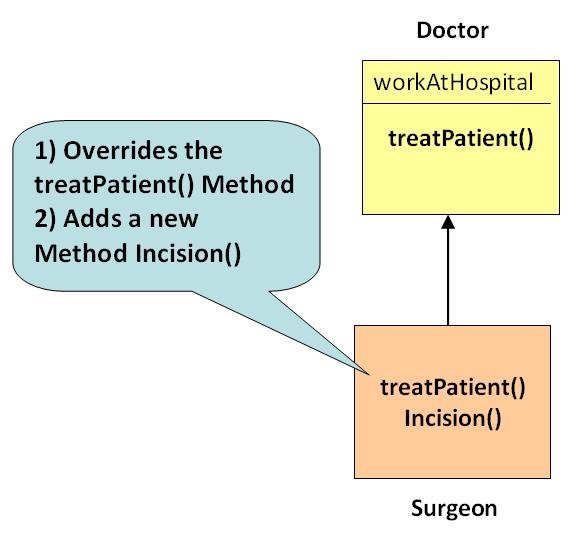


## Method Overriding in Java

Method Overriding is redefining a super class method in a sub class.

### Rules for Method Overriding

* The method signature i.e. method name, parameter list and return type have to match exactly.
* The overridden method can widen the accessibility but not narrow it, i.e. if it is private in the base class, the child class can make it public but not vice versa.



**Example**

class Doctor{

public void treatPatient(){

// treatPatient method

}

class Surgeon extends Doctor{

public void treatPatient(){

// treatPatient method

}

}

Class run{

public static void main (String args[]){

Doctor doctorObj = new Doctor()

// treatPatient method in class Doctor will be executed

doctorObj.treatPatient();

Surgeon surgeonObj = new Surgeon();

// treatPatient method in class Surgeon will be executed

surgeonObj.treatPatient();

}

}

## Difference between Overloading and Overriding

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is in the same class, where more than 1 1 method have the same name but different signatures. | Method overriding is when one of the  methods in the super class is  redefined in the sub-class. In this case, the signature of the method remains the same. |
| **Ex:**  void sum (int a , int b);  void sum (int a , int b, int c);  void sum (float a, double b); | **Ex:**  class X{  public int sum(){  // some code  }  }  class Y extends X{  public int sum(){  //overridden method  //signature is same  }  } |

## What is Dynamic Polymorphism?

Dynamic Polymorphism in OOPs is the mechanism by which multiple methods can be defined with same name and signature in the superclass and subclass. The call to an overridden method are resolved at run time.

### Dynamic Polymorphism Example:

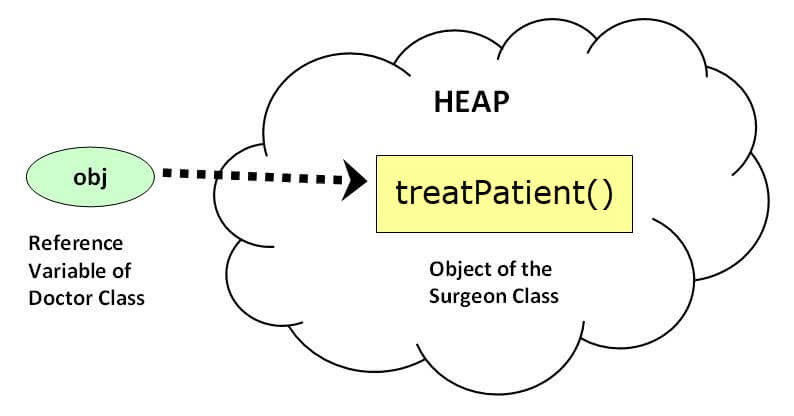
A reference variable of the super class can refer to a sub class object

Doctor obj = new Surgeon();

Consider the statement

obj.treatPatient();

Here the reference variable “obj” is of the parent class, but the object it is pointing to is of the child class (as shown in the below diagram example of Polymorphism).



obj.treatPatient() will execute treatPatient() method of the sub-class – Surgeon

If a base class reference is used to call a method, the method to be invoked is decided by the JVM, depending on the object the reference is pointing to

For example, even though obj is a reference to Doctor, it calls the method of Surgeon, as it points to a Surgeon object

This is decided during run-time and hence termed **dynamic**or **run-time polymorphism**

## What is encapsulation:

**Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.



We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

However if we setup public getter and setter methods to update (for example void setSSN(int ssn))and read (for example  int getSSN()) the private data fields then the outside class can access those private data fields via public methods.

This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That’s why encapsulation is known as **data hiding.**Lets see an example to understand this concept better.

## Example of Encapsulation in Java

How to implement encapsulation in java:  
1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.  
2) Have getter and setter methods in the class to set and get the values of the fields.

class EncapsulationDemo{

private int ssn;

private String empName;

private int empAge;

//Getter and Setter methods

public int getEmpSSN(){

return ssn;

}

public String getEmpName(){

return empName;

}

public int getEmpAge(){

return empAge;

}

public void setEmpAge(int newValue){

empAge = newValue;

}

public void setEmpName(String newValue){

empName = newValue;

}

public void setEmpSSN(int newValue){

ssn = newValue;

}

}

public class EncapsTest{

public static void main(String args[]){

EncapsulationDemo obj = new EncapsulationDemo();

obj.setEmpName("Mario");

obj.setEmpAge(32);

obj.setEmpSSN(112233);

System.out.println("Employee Name: " + obj.getEmpName());

System.out.println("Employee SSN: " + obj.getEmpSSN());

System.out.println("Employee Age: " + obj.getEmpAge());

}

}

**Output:**

Employee Name: Mario

Employee SSN: 112233

Employee Age: 32

In above example all the three data members (or data fields) are private(see: [**Access Modifiers in Java**](https://beginnersbook.com/2013/05/java-access-modifiers/)) which cannot be accessed directly. These fields can be accessed via public methods only. Fields empName, ssn and empAge are made hidden data fields using encapsulation technique of OOPs.

## Advantages of encapsulation

1. It improves maintainability and flexibility and re-usability: for e.g. In the above code the implementation code of void setEmpName(String name) and String getEmpName() can be changed at any point of time. Since the implementation is purely hidden for outside classes they would still be accessing the private field empName using the same methods (setEmpName(String name) and getEmpName()). Hence the code can be maintained at any point of time without breaking the classes that uses the code. This improves the re-usability of the underlying class.
2. The fields can be made read-only (If we don’t define setter methods in the class) or write-only (If we don’t define the getter methods in the class). For e.g. If we have a field(or variable) that we don’t want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.
3. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.

Encapsulation is also known as “**data Hiding**“.