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.



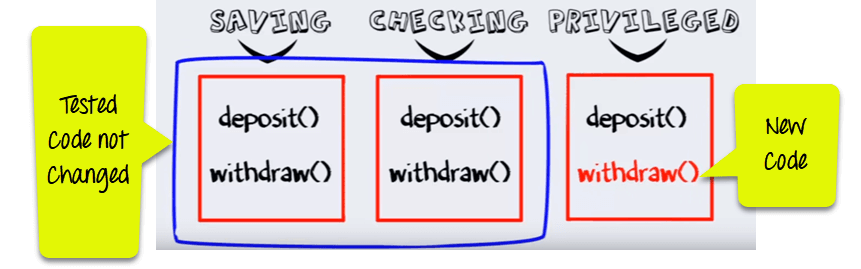
Java Polymorphism Example

### Change in Software Requirement

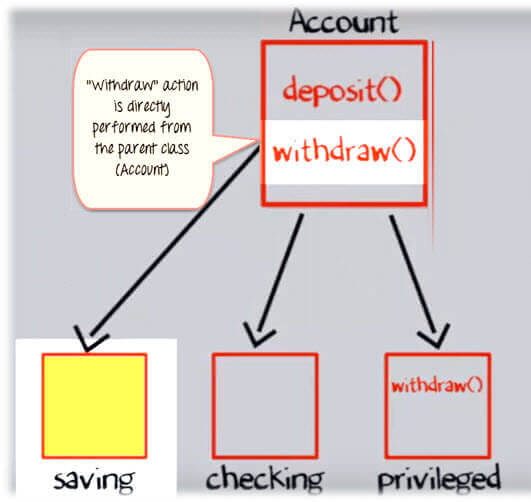
There is a change in the requirement specification, something that is so common in the software industry. You are supposed to add functionality privileged Banking Account with Overdraft Facility.

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

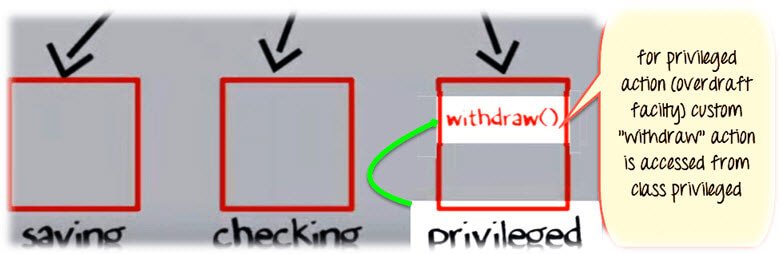
So, withdraw method for privileged needs to implemented afresh. But you do not change the tested piece of code in Savings and Checking account. This is advantage of OOPS



**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.**

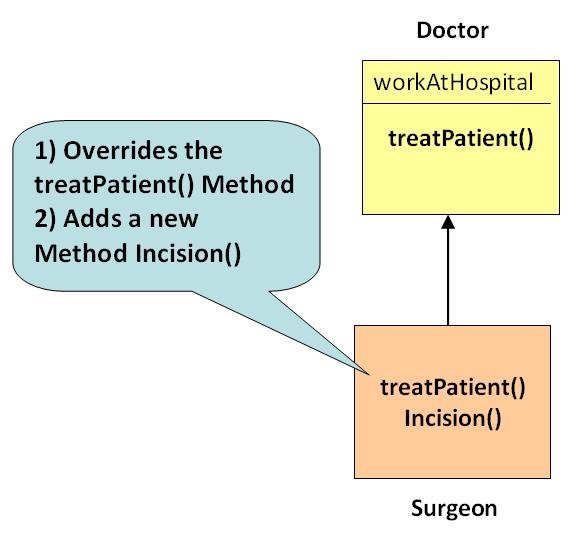


## Method Overriding in Java

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:

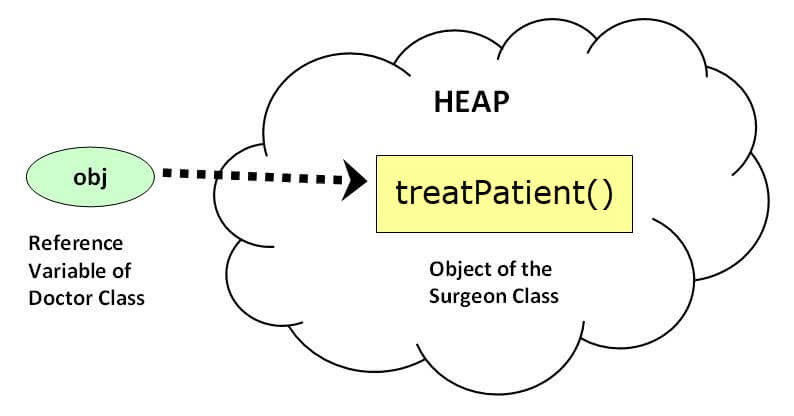
A reference variable of the super class can refer to a sub class object

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**“.

# **Super Keyword in Java**

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.



## 1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

1. **class** Animal{
2. String color="white";
3. }
4. **class** Dog **extends** Animal{
5. String color="black";
6. **void** printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(**super**.color);//prints color of Animal class
9. }
10. }
11. **class** TestSuper1{
12. **public** **static** **void** main(String args[]){
13. Dog d=**new** Dog();
14. d.printColor();
15. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestSuper1" \t "_blank)**

Output:

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

black

white

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

## 2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("eating bread...");}
6. **void** bark(){System.out.println("barking...");}
7. **void** work(){
8. **super**.eat();
9. bark();
10. }
11. }
12. **class** TestSuper2{
13. **public** **static** **void** main(String args[]){
14. Dog d=**new** Dog();
15. d.work();
16. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestSuper2" \t "_blank)**

Output:

eating...

barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

## 3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. **super**();
7. System.out.println("dog is created");
8. }
9. }
10. **class** TestSuper3{
11. **public** **static** **void** main(String args[]){
12. Dog d=**new** Dog();
13. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestSuper3" \t "_blank)**

Output:

animal is created

dog is created

#### **Note: super() is added in each class constructor automatically by compiler if there is no super() or this().**



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

**Another example of super keyword where super() is provided by the compiler implicitly.**

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. System.out.println("dog is created");
7. }
8. }
9. **class** TestSuper4{
10. **public** **static** **void** main(String args[]){
11. Dog d=**new** Dog();
12. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestSuper4" \t "_blank)**

Output:

animal is created

dog is created

## super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

1. **class** Person{
2. **int** id;
3. String name;
4. Person(**int** id,String name){
5. **this**.id=id;
6. **this**.name=name;
7. }
8. }
9. **class** Emp **extends** Person{
10. **float** salary;
11. Emp(**int** id,String name,**float** salary){
12. **super**(id,name);//reusing parent constructor
13. **this**.salary=salary;
14. }
15. **void** display(){System.out.println(id+" "+name+" "+salary);}
16. }
17. **class** TestSuper5{
18. **public** **static** **void** main(String[] args){
19. Emp e1=**new** Emp(1,"ankit",45000f);
20. e1.display();
21. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestSuper5" \t "_blank)**

Output:

1 ankit 45000

# **Java this keyword:**

## Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. [this can be used to refer current class instance variable.](https://www.javatpoint.com/this1)
2. [this can be used to invoke current class method (implicitly)](https://www.javatpoint.com/this2)
3. [this() can be used to invoke current class constructor.](https://www.javatpoint.com/this3)
4. [this can be passed as an argument in the method call.](https://www.javatpoint.com/this4)
5. [this can be passed as argument in the constructor call.](https://www.javatpoint.com/this5)
6. [this can be used to return the current class instance from the method.](https://www.javatpoint.com/this6)

**Suggestion:** If you are beginner to java, lookup only three usages of this keyword.



### **1) this: to refer current class instance variable**

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

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

#### **Understanding the problem without this keyword**

Let's understand the problem if we don't use this keyword by the example given below:

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. **void** display(){
11. System.out.println(rollno+" "+name+" "+fee);
12. }
13. }
14. **class** TestThis1{
15. **public** **static** **void** main(String args[]){
16. Student s1=**new** Student(111,"ankit",5000f);
17. Student s2=**new** Student(112,"sumit",6000f);
18. s1.display();
19. s2.display();
20. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis1)

**Output:**

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

#### **Solution of the above problem by this keyword**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis2{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis2)

**Output:**

111 ankit 5000.0

112 sumit 6000.0

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

#### **Program where this keyword is not required**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** r,String n,**float** f){
6. rollno=r;
7. name=n;
8. fee=f;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis3{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis3)

**Output:**

111 ankit 5000.0

112 sumit 6000.0

#### **It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.**

### **2) this: to invoke current class method**

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis4{
10. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis4)

**Output:**

hello n

hello m

### **3) this() : to invoke current class constructor**

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

1. **class** A{
2. A(){System.out.println("hello a");}
3. A(**int** x){
4. **this**();
5. System.out.println(x);
6. }
7. }
8. **class** TestThis5{
9. **public** **static** **void** main(String args[]){
10. A a=**new** A(10);
11. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis5)

**Output:**

hello a

10

**Calling parameterized constructor from default constructor:**

1. **class** A{
2. A(){
3. **this**(5);
4. System.out.println("hello a");
5. }
6. A(**int** x){
7. System.out.println(x);
8. }
9. }
10. **class** TestThis6{
11. **public** **static** **void** main(String args[]){
12. A a=**new** A();
13. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis6)

**Output:**

5

hello a

### **Real usage of this() constructor call**

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**(rollno,name,course);//reusing constructor
12. **this**.fee=fee;
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis7{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis7)

**Output:**

111 ankit java 0.0

112 sumit java 6000.0

#### **Rule: Call to this() must be the first statement in constructor.**

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**.fee=fee;
12. **this**(rollno,name,course);//C.T.Error
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis8{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestThis8)

**Output:**

Compile Time Error: Call to this must be first statement in constructor

# Java final keyword

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

* the final variable cannot be reinitialized with another value
* the final method cannot be overridden
* the final class cannot be extended

## 1. Java final Variable

In Java, we cannot change the value of a final variable. For example,

class Main {

public static void main(String[] args) {

// create a final variable

final int AGE = 32;

// try to change the final variable

AGE = 45;

System.out.println("Age: " + AGE);

}

}

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

In the above program, we have created a final variable named age. And we have tried to change the value of the final variable.

When we run the program, we will get a compilation error with the following message.

cannot assign a value to final variable AGE

AGE = 45;

^

**Note**: It is recommended to use uppercase to declare final variables in Java.

## 2. Java final Method

Before you learn about final methods and final classes, make sure you know about the [Java Inheritance](https://www.programiz.com/java-programming/inheritance).

In Java, the final method cannot be overridden by the child class. For example,

class FinalDemo {

// create a final method

public final void display() {

System.out.println("This is a final method.");

}

}

class Main extends FinalDemo {

// try to override final method

public final void display() {

System.out.println("The final method is overridden.");

}

public static void main(String[] args) {

Main obj = new Main();

obj.display();

}

}

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

In the above example, we have created a final method named display() inside the FinalDemo class. Here, the Main class inherits the FinalDemo class.

We have tried to override the final method in the Main class. When we run the program, we will get a compilation error with the following message.

display() in Main cannot override display() in FinalDemo

public final void display() {

^

overridden method is final

## 3. Java final Class

In Java, the final class cannot be inherited by another class. For example,

// create a final class

final class FinalClass {

public void display() {

System.out.println("This is a final method.");

}

}

// try to extend the final class

class Main extends FinalClass {

public void display() {

System.out.println("The final method is overridden.");

}

public static void main(String[] args) {

Main obj = new Main();

obj.display();

}

}

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

In the above example, we have created a final class named FinalClass. Here, we have tried to inherit the final class by the Main class.

When we run the program, we will get a compilation error with the following message.

cannot inherit from final FinalClass

class Main extends FinalClass {

^

## IS-A Relationship (Inheritance) and has-a relationship

**IS-A Relationship (Inheritance)**

IS-A Relationship can be formed with a minimum of two classes with a relation. This is referred as Inheritance in java.  
Let us take a simple example for our understanding. We know every four-wheeler Car is a Vehicle.  
Honda is a Car  
Suziki is a Car  
Toyota is a Car  
benz is a Car  
  
So, From these examples all four wheeler brands are Cars and all have some common features such as color, max speed, gears type etc. These common features will be in Car class and car specific features will be in Car Brand car.  
  
IS-A relationship is implemented by using the "extends" keyword in java inheritance. Whenever you find the "extends" keyword anywhere in your application that is IS-A Relationship.  
  
Creating one is Car class and many car specific classes.

**IS-A Relationship Example**

// Common features class

class Car {

int avgSpeed;

int currentSpeed;

public int getCurrentSpeed() {

return currentSpeed;

}

public void setCurrentSpeed(int currentSpeed) {

this.currentSpeed = currentSpeed;

}

public int getAvgSpeed() {

return avgSpeed;

}

public void setAvgSpeed(int maxSpeed) {

this.avgSpeed = maxSpeed;

}

public int getNoOfGears() {

return 6;

}

public String getDefaultColor() {

return "White";

}

}

In-Car class, it has common methods which are used by all its subclasses. It has default no of gears, colour, can be set the current speed.  
  
Below is a Honda car which is a subclass of Car. It has its own specific max speed is 120. Inside the Honda car, we can use all methods of Car class.

// Honda specific implementation class

class Honda extends Car {

public void startHondaCar() {

setAvgSpeed(70);

System.out.println("Started Honda Car with average speed : " + getAvgSpeed());

}

public int getMaxSpeed() {

return 120;

}

public void stopHondaCar() {

System.out.println("Honda Car has stopped");

}

}

Below is a Maruthi car which is a subclass of Car. It has its own specific max speed is 100. Inside the Honda car, we can use all methods of Car class.

// Maruthi specific implementation class

class Maruthi extends Car {

public void startHondaCar() {

setAvgSpeed(100);

System.out.println("Started Maruthi Car with average speed : " + getAvgSpeed());

}

public int getMaxSpeed() {

return 100;

}

public void stopHondaCar() {

System.out.println("Maruthi Car has stopped");

}

}

In these two Honda, Maruthi classes have access to Car class and all Car methods are accessible from these subclasses.  
  
**Main class of IS-A relationship.**

// Main method for IS-A Relationship

public class IsARelationShip {

public static void main(String[] args) {

System.out.println("Honda specific implementation.");

Honda hondaCar = new Honda();

hondaCar.startHondaCar();

System.out.println("Honda car max speed: " + hondaCar.getMaxSpeed());

hondaCar.stopHondaCar();

System.out.println("\nMaruthi specific implementation.");

Maruthi maruthiCar = new Maruthi();

maruthiCar.startHondaCar();

System.out.println("Honda car max speed: " + maruthiCar.getMaxSpeed());

maruthiCar.stopHondaCar();

}

}

**Output:**

Honda specific implementation.

Started Honda Car with average speed : 70

Honda car max speed: 120

Honda Car has stopped

Maruthi specific implementation.

Started Maruthi Car with average speed : 100

Honda car max speed: 100

Maruthi Car has stopped

In this example program, Car class has all the common features of a car which all are required by all its implementation cars which can be default features. If Honda wants its own specific then they have the ability to provide its own implementation for those methods which is called as Method Overriding.  
  
Please go through the above program and leave your questions in the comments section.

**HAS-A Relationship (Association)**

HAS-A relationship also can be formed by a minimum of two classes. If any class has an instance variable that referring another class.

Let us break them down into examples for better understanding. Honda car has an Engine, Maruthi car has an Engine.

**HAS-A Relationship Example**

Creating an Engine class that has engine related functions.

// Engine specific implementation class

class Engine {

public void startWithDieselEngine() {

System.out.println("Diesel engine has added and it's started");

}

public void startWithPetrolEngine() {

System.out.println("Petrol engine has added and it's started");

}

public void startWithAutomaticEngine() {

System.out.println("Automatic engine has added and it's started");

}

}

Creating ToyotaDiesel class which has instance variable of Engine class. Instance variable engine calls startWithDieselEngine() method and ignore the remaining methods in the Engine class.

// Toyota Diesel car

class ToyotaDiesel {

public Engine engine;

public Engine getEngine() {

return engine;

}

public void setEngine(Engine engine) {

this.engine = engine;

}

public ToyotaDiesel() {

System.out.println("Toyota Diesel car has constructed");

}

public void startEngine() {

engine.startWithDieselEngine();

}

}

Another class, instance variable referring to Engine class. Invoking benz automatic engine startWithDieselEngine().

// Benz Automatic car

class BenzAutomatic {

public Engine engine;

public Engine getEngine() {

return engine;

}

public void setEngine(Engine engine) {

this.engine = engine;

}

public BenzAutomatic() {

System.out.println("Benz Automatic car has constructed");

}

public void startEngine() {

engine.startWithDieselEngine();

}

}

Main method IS-A relationship class:

// Main method for IS-A Relationship

public class HasARelationShip {

public static void main(String[] args) {

// Creating Engine Object

Engine engine = new Engine();

// Toyota diesel car

System.out.println("ToyotaDiesel specific implementation.");

ToyotaDiesel toyotaDieselCar = new ToyotaDiesel();

toyotaDieselCar.setEngine(engine);

toyotaDieselCar.startEngine();

// Benz Automatic car

System.out.println("\nBenzAutomatic specific implementation.");

BenzAutomatic benzAutomatic = new BenzAutomatic();

benzAutomatic.setEngine(engine);

benzAutomatic.startEngine();

}

}

**Output:**

ToyotaDiesel specific implementation.

Toyota Diesel car has constructed

Diesel engine has added and it's started

BenzAutomatic specific implementation.

Benz Automatic car has constructed

Diesel engine has added and it's started

Object Class:

# **Object class in Java**

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

1. Object obj=getObject();//we don't know what object will be returned from this method

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

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



### **Methods of Object class**

|  |
| --- |
| The Object class provides many methods. They are as follows: |

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

String:

### **What is String in Java?**

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

### **How to create a string object?**

There are two ways to create String object:

1. By string literal
2. By new keyword

### **1) String Literal**

Java String literal is created by using double quotes. For Example:

1. String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

#### **Note: String objects are stored in a special memory area known as the "string constant pool".**

### **Why Java uses the concept of String literal?**

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

### **2) By new keyword**

1. String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine)

will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

### **Java String Example**

**StringExample.java**

1. **public** **class** StringExample{
2. **public** **static** **void** main(String args[]){
3. String s1="java";//creating string by Java string literal
4. **char** ch[]={'s','t','r','i','n','g','s'};
5. String s2=**new** String(ch);//converting char array to string
6. String s3=**new** String("example");//creating Java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=StringExample" \t "_blank)**

**Output:**

java

strings

example

The above code, converts a **char** array into a **String** object. And displays the String objects **s1, s2**, and **s3** on console using **println()** method.

### **Java String class methods**

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | It returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | It returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns a formatted string. |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | It returns formatted string with given locale. |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index. |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | It returns substring for given begin index and end index. |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | It returns true or false after matching the sequence of char value. |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | It returns a joined string. |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | It checks the equality of string with the given object. |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | It checks if string is empty. |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | It concatenates the specified string. |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified char value. |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | It replaces all occurrences of the specified CharSequence. |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | It compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex. |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | It returns a split string matching regex and limit. |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | It returns an interned string. |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index. |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified char value index starting with given index. |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index. |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | It returns the specified substring index starting with given index. |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | It returns a string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | It returns a string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | It removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | It converts given type into string. It is an overloaded method. |

Let's use some important methods of String class.

### **Java String toUpperCase() and toLowerCase() method**

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

63.5M

1.3K

Features of Java - Javatpoint

**Next**

**Stay**

**Stringoperation1.java**

1. **public** **class** Stringoperation1
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. String s="Sachin";
6. System.out.println(s.toUpperCase());//SACHIN
7. System.out.println(s.toLowerCase());//sachin
8. System.out.println(s);//Sachin(no change in original)
9. }
10. }

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

**Output:**

SACHIN

sachin

Sachin

### **Java String trim() method**

The String class trim() method eliminates white spaces before and after the String.

**Stringoperation2.java**

1. **public** **class** Stringoperation2
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. String s="  Sachin  ";
6. System.out.println(s);//  Sachin
7. System.out.println(s.trim());//Sachin
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass1)

**Output:**

Sachin

Sachin

### **Java String startsWith() and endsWith() method**

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

**Stringoperation3.java**

1. **public** **class** Stringoperation3
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. String s="Sachin";
6. System.out.println(s.startsWith("Sa"));//true
7. System.out.println(s.endsWith("n"));//true
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass2)

**Output:**

true

true

### **Java String charAt() Method**

The String class charAt() method returns a character at specified index.

**Stringoperation4.java**

1. **public** **class** Stringoperation4
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. String s="Sachin";
6. System.out.println(s.charAt(0));//S
7. System.out.println(s.charAt(3));//h
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass3)

**Output:**

S

h

### **Java String length() Method**

The String class length() method returns length of the specified String.

**Stringoperation5.java**

1. **public** **class** Stringoperation5
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. String s="Sachin";
6. System.out.println(s.length());//6
7. }
8. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass4)

**Output:**

6

### **Java String intern() Method**

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a String equal to this String object as determined by the equals(Object) method, then the String from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

**Stringoperation6.java**

1. **public** **class** Stringoperation6
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. String s=**new** String("Sachin");
6. String s2=s.intern();
7. System.out.println(s2);//Sachin
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testmethodofstringclass5)

**Output:**

Sachin

### **Java String valueOf() Method**

The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.

**Stringoperation7.java**

1. **public** **class** Stringoperation7
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. **int** a=10;
6. String s=String.valueOf(a);
7. System.out.println(s+10);
8. }
9. }

**Output:**

1010

### **Java String replace() Method**

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

**Stringoperation8.java**

1. **public** **class** Stringoperation8
2. {
3. **public** **static** **void** main(String ar[])
4. {
5. String s1="Java is a programming language. Java is a platform. Java is an Island.";
6. String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
7. System.out.println(replaceString);
8. }
9. }

**Output:**

Kava is a programming language. Kava is a platform. Kava is an Island.

# **Immutable String in Java:**

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.

Let's try to understand the concept of immutability by the example given below:

**Testimmutablestring.java**

3M

The Laowa 100mm Macro lens has ZERO FLARE!

1. **class** Testimmutablestring{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s.concat(" Tendulkar");//concat() method appends the string at the end
5. System.out.println(s);//will print Sachin because strings are immutable objects
6. }
7. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring" \t "_blank)**

**Output:**

Sachin

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



As you can see in the above figure that two objects are created but **s** reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

**Testimmutablestring1.java**

1. **class** Testimmutablestring1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin";
4. s=s.concat(" Tendulkar");
5. System.out.println(s);
6. }
7. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring1" \t "_blank)**

**Output:**

Sachin Tendulkar

In such a case, s points to the "Sachin Tendulkar". Please notice that still Sachin object is not modified.

### **Why String objects are immutable in Java?**

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Following are some features of String which makes String objects immutable.

# **Java String compare**

We can compare String in Java on the basis of content and reference.

1. By Using equals() Method
2. By Using == Operator

## 1) By Using equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

* **public boolean equals(Object another)** compares this string to the specified object.
* **public boolean equalsIgnoreCase(String another)** compares this string to another string, ignoring case.

**Teststringcomparison1.java**

1. **class** Teststringcomparison1{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. String s4="Saurav";
7. System.out.println(s1.equals(s2));//true
8. System.out.println(s1.equals(s3));//true
9. System.out.println(s1.equals(s4));//false
10. }
11. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison1)

**Output:**

true

true

false

In the above code, two strings are compared using **equals()** method of **String** class. And the result is printed as boolean values, **true** or **false**.

**Teststringcomparison2.java**

1. **class** Teststringcomparison2{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="SACHIN";
6. System.out.println(s1.equals(s2));//false
7. System.out.println(s1.equalsIgnoreCase(s2));//true
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison2)

**Output:**

false

true

In the above program, the methods of **String** class are used. The **equals()** method returns true if String objects are matching and both strings are of same case. **equalsIgnoreCase()** returns true regardless of cases of strings.

[Click here for more about equals() method](https://www.javatpoint.com/java-string-equals)

## 2) By Using == operator

The == operator compares references not values.

**Teststringcomparison3.java**

1. **class** Teststringcomparison3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. System.out.println(s1==s2);//true (because both refer to same instance)
7. System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8. }
9. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison3)

**Output:**

true

false

# **Substring in Java**

A part of String is called **substring**. In other words, substring is a subset of another String. Java String class provides the built-in substring() method that extract a substring from the given string by using the index values passed as an argument. In case of substring() method startIndex is inclusive and endIndex is exclusive.

Suppose the string is "**computer**", then the substring will be com, compu, ter, etc.

#### **Note: Index starts from 0.**

You can get substring from the given String object by one of the two methods:

1. **public String substring(int startIndex):**  
   This method returns new String object containing the substring of the given string from specified startIndex (inclusive). The method throws an IndexOutOfBoundException when the startIndex is larger than the length of String or less than zero.
2. **public String substring(int startIndex, int endIndex):**  
   This method returns new String object containing the substring of the given string from specified startIndex to endIndex. The method throws an IndexOutOfBoundException when the startIndex is less than zero or startIndex is greater than endIndex or endIndex is greater than length of String.

In case of String:

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

* **startIndex:** inclusive
* **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

1. String s="hello";
2. System.out.println(s.substring(0,2)); //returns he  as a substring

In the above substring, 0 points the first letter and 2 points the second letter i.e., e (because end index is exclusive).

### **Example of Java substring() method**

**TestSubstring.java**

1. **public** **class** TestSubstring{
2. **public** **static** **void** main(String args[]){
3. String s="SachinTendulkar";
4. System.out.println("Original String: " + s);
5. System.out.println("Substring starting from index 6: " +s.substring(6));//Tendulkar
6. System.out.println("Substring starting from index 0 to 6: "+s.substring(0,6)); //Sachin
7. }
8. }

**Output:**

Original String: SachinTendulkar

Substring starting from index 6: Tendulkar

Substring starting from index 0 to 6: Sachin

The above [Java programs](https://www.javatpoint.com/java-programs), demonstrates variants of the **substring()** method of **String** class. The startindex is inclusive and endindex is exclusive.

### **Using String.split() method:**

The split() method of String class can be used to extract a substring from a sentence. It accepts arguments in the form of a regular expression.

**TestSubstring2.java**

1. **import** java.util.\*;
3. **public** **class** TestSubstring2
4. {
5. /\* Driver Code \*/
6. **public** **static** **void** main(String args[])
7. {
8. String text= **new** String("Hello, My name is Sachin");
9. /\* Splits the sentence by the delimeter passed as an argument \*/
10. String[] sentences = text.split("\\.");
11. System.out.println(Arrays.toString(sentences));
12. }
13. }

**Output:**

[Hello, My name is Sachin]

In the above program, we have used the split() method. It accepts an argument \\. that checks a in the sentence and splits the string into another string. It is stored in an array of String objects sentences.

# **Java StringBuffer Class**

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

#### **Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.**

### **Important Constructors of StringBuffer Class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuffer() | It creates an empty String buffer with the initial capacity of 16. |
| StringBuffer(String str) | It creates a String buffer with the specified string.. |
| StringBuffer(int capacity) | It creates an empty String buffer with the specified capacity as length. |

### **Important methods of StringBuffer class**

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| public synchronized StringBuffer | append(String s) | It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public synchronized StringBuffer | insert(int offset, String s) | It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public synchronized StringBuffer | replace(int startIndex, int endIndex, String str) | It is used to replace the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | delete(int startIndex, int endIndex) | It is used to delete the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | reverse() | is used to reverse the string. |
| public int | capacity() | It is used to return the current capacity. |
| public void | ensureCapacity(int minimumCapacity) | It is used to ensure the capacity at least equal to the given minimum. |
| public char | charAt(int index) | It is used to return the character at the specified position. |
| public int | length() | It is used to return the length of the string i.e. total number of characters. |
| public String | substring(int beginIndex) | It is used to return the substring from the specified beginIndex. |
| public String | substring(int beginIndex, int endIndex) | It is used to return the substring from the specified beginIndex and endIndex. |

# **Java StringBuilder Class**

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

## Important Constructors of StringBuilder class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuilder() | It creates an empty String Builder with the initial capacity of 16. |
| StringBuilder(String str) | It creates a String Builder with the specified string. |
| StringBuilder(int length) | It creates an empty String Builder with the specified capacity as length. |

## Important methods of StringBuilder class

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public StringBuilder insert(int offset, String s) | It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public StringBuilder replace(int startIndex, int endIndex, String str) | It is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | It is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | It is used to reverse the string. |
| public int capacity() | It is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | It is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | It is used to return the character at the specified position. |
| public int length() | It is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | It is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | It is used to return the substring from the specified beginIndex and endIndex. |

# **How to create Immutable class?**

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

### **Example to create Immutable class**

In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method.

**ImmutableDemo.java**

1. **public** **final** **class** Employee
2. {
3. **final** String pancardNumber;
4. **public** Employee(String pancardNumber)
5. {
6. **this**.pancardNumber=pancardNumber;
7. }
8. **public** String getPancardNumber(){
9. **return** pancardNumber;
10. }
11. }
12. **public** **class** ImmutableDemo
13. {
14. **public** **static** **void** main(String ar[])
15. {
16. Employee e = **new** Employee("ABC123");
17. String s1 = e.getPancardNumber();
18. System.out.println("Pancard Number: " + s1);
19. }
20. }

**Output:**

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

Pancard Number: ABC123

The above class is immutable because:

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.

These points makes this class as immutable.

# **Java toString() Method**

If you want to represent any object as a string, **toString() method** comes into existence.

The toString() method returns the String representation of the object.

If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

### **Advantage of Java toString() method**

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

3.2M

### **Understanding problem without toString() method**

Let's see the simple code that prints reference.

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** **static** **void** main(String args[]){
13. Student s1=**new** Student(101,"Raj","lucknow");
14. Student s2=**new** Student(102,"Vijay","ghaziabad");
16. System.out.println(s1);//compiler writes here s1.toString()
17. System.out.println(s2);//compiler writes here s2.toString()
18. }
19. }

**Output:**

Student@1fee6fc

Student@1eed786

As you can see in the above example, printing s1 and s2 prints the hashcode values of the objects but I want to print the values of these objects. Since Java compiler internally calls toString() method, overriding this method will return the specified values. Let's understand it with the example given below:

## Example of Java toString() method

Let's see an example of toString() method.

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. String city;
6. Student(**int** rollno, String name, String city){
7. **this**.rollno=rollno;
8. **this**.name=name;
9. **this**.city=city;
10. }
12. **public** String toString(){//overriding the toString() method
13. **return** rollno+" "+name+" "+city;
14. }
15. **public** **static** **void** main(String args[]){
16. Student s1=**new** Student(101,"Raj","lucknow");
17. Student s2=**new** Student(102,"Vijay","ghaziabad");
19. System.out.println(s1);//compiler writes here s1.toString()
20. System.out.println(s2);//compiler writes here s2.toString()
21. }
22. }

**Output:**

101 Raj lucknow

102 Vijay ghaziabad

In the above program, Java compiler internally calls **toString()** method, overriding this method will return the specified values of **s1** and **s2** objects of Student class.

# **Java Regex:**

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

The **Java Regex** or Regular Expression is an API to define a pattern for searching or manipulating strings.

It is widely used to define the constraint on strings such as password and email validation. After learning Java regex tutorial, you will be able to test your regular expressions by the Java Regex Tester Tool.

Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.

#### **java.util.regex package**

The Matcher and Pattern classes provide the facility of Java regular expression. The java.util.regex package provides following classes and interfaces for regular expressions.

64.7M

1.2K

Exception Handling in Java - Javatpoint

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class



## Matcher class

It implements the **MatchResult** interface. It is a regex engine which is used to perform match operations on a character sequence.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | boolean matches() | test whether the regular expression matches the pattern. |
| 2 | boolean find() | finds the next expression that matches the pattern. |
| 3 | boolean find(int start) | finds the next expression that matches the pattern from the given start number. |
| 4 | String group() | returns the matched subsequence. |
| 5 | int start() | returns the starting index of the matched subsequence. |
| 6 | int end() | returns the ending index of the matched subsequence. |
| 7 | int groupCount() | returns the total number of the matched subsequence. |

## Pattern class

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | static Pattern compile(String regex) | compiles the given regex and returns the instance of the Pattern. |
| 2 | Matcher matcher(CharSequence input) | creates a matcher that matches the given input with the pattern. |
| 3 | static boolean matches(String regex, CharSequence input) | It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern. |
| 4 | String[] split(CharSequence input) | splits the given input string around matches of given pattern. |
| 5 | String pattern() | returns the regex pattern. |

### **Example of Java Regular Expressions**

There are three ways to write the regex example in Java.

1. **import** java.util.regex.\*;
2. **public** **class** RegexExample1{
3. **public** **static** **void** main(String args[]){
4. //1st way
5. Pattern p = Pattern.compile(".s");//. represents single character
6. Matcher m = p.matcher("as");
7. **boolean** b = m.matches();
9. //2nd way
10. **boolean** b2=Pattern.compile(".s").matcher("as").matches();
12. //3rd way
13. **boolean** b3 = Pattern.matches(".s", "as");
15. System.out.println(b+" "+b2+" "+b3);
16. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample1" \t "_blank)**

#### **Output**

true true true

## Regular Expression . Example

The . (dot) represents a single character.

1. **import** java.util.regex.\*;
2. **class** RegexExample2{
3. **public** **static** **void** main(String args[]){
4. System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)
5. System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)
6. System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)
7. System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)
8. System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)
9. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample2" \t "_blank)**

## Regex Character classes

|  |  |  |
| --- | --- | --- |
| **No.** | **Character Class** | **Description** |
| 1 | [abc] | a, b, or c (simple class) |
| 2 | [^abc] | Any character except a, b, or c (negation) |
| 3 | [a-zA-Z] | a through z or A through Z, inclusive (range) |
| 4 | [a-d[m-p]] | a through d, or m through p: [a-dm-p] (union) |
| 5 | [a-z&&[def]] | d, e, or f (intersection) |
| 6 | [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| 7 | [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) |

### **Example of Java Regular Expressions**

1. **import** java.util.regex.\*;
2. **class** RegexExample3{
3. **public** **static** **void** main(String args[]){
4. System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
5. System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)
6. System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)
7. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample3" \t "_blank)**

## Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| X? | X occurs once or not at all |
| X+ | X occurs once or more times |
| X\* | X occurs zero or more times |
| X{n} | X occurs n times only |
| X{n,} | X occurs n or more times |
| X{y,z} | X occurs at least y times but less than z times |

## Regular Expression Character classes and Quantifiers Example

1. **import** java.util.regex.\*;
2. **class** RegexExample4{
3. **public** **static** **void** main(String args[]){
4. System.out.println("? quantifier ....");
5. System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)
6. System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)
7. System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)
8. System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)
9. System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)
11. System.out.println("+ quantifier ....");
12. System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)
13. System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)
14. System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once)
15. System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern)
17. System.out.println("\* quantifier ....");
18. System.out.println(Pattern.matches("[amn]\*", "ammmna"));//true (a or m or n may come zero or more times)
20. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample4" \t "_blank)**

## Regex Metacharacters

The regular expression metacharacters work as shortcodes.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| . | Any character (may or may not match terminator) |
| \d | Any digits, short of [0-9] |
| \D | Any non-digit, short for [^0-9] |
| \s | Any whitespace character, short for [\t\n\x0B\f\r] |
| \S | Any non-whitespace character, short for [^\s] |
| \w | Any word character, short for [a-zA-Z\_0-9] |
| \W | Any non-word character, short for [^\w] |
| \b | A word boundary |
| \B | A non word boundary |

## Regular Expression Metacharacters Example

1. **import** java.util.regex.\*;
2. **class** RegexExample5{
3. **public** **static** **void** main(String args[]){
4. System.out.println("metacharacters d....");\\d means digit
6. System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)
7. System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)
8. System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes more than once)
9. System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)
11. System.out.println("metacharacters D....");\\D means non-digit
13. System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes more than once)
14. System.out.println(Pattern.matches("\\D", "1"));//false (digit)
15. System.out.println(Pattern.matches("\\D", "4443"));//false (digit)
16. System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)
17. System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes once)
19. System.out.println("metacharacters D with quantifier....");
20. System.out.println(Pattern.matches("\\D\*", "mak"));//true (non-digit and may come 0 or more times)
22. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample5" \t "_blank)**

## Regular Expression Question 1

1. /\*Create a regular expression that accepts alphanumeric characters only.
2. Its length must be six characters long only.\*/
4. **import** java.util.regex.\*;
5. **class** RegexExample6{
6. **public** **static** **void** main(String args[]){
7. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true
8. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false (more than 6 char)
9. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true
10. System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is not matched)
11. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample6" \t "_blank)**

## Regular Expression Question 2

1. /\*Create a regular expression that accepts 10 digit numeric characters
2. starting with 7, 8 or 9 only.\*/
4. **import** java.util.regex.\*;
5. **class** RegexExample7{
6. **public** **static** **void** main(String args[]){
7. System.out.println("by character classes and quantifiers ...");
8. System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true
9. System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true
11. System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)
12. System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)
13. System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true
15. System.out.println("by metacharacters ...");
16. System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true
17. System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)
19. }}

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=RegexExample7" \t "_blank)**

Output:

Enter regex pattern: java

Enter text: this is java, do you know java

I found the text java starting at index 8 and ending at index 12

I found the text java starting at index 26 and ending at index 30

# **Exception Handling in Java:**

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

## What is Exception in Java?

**Dictionary Meaning:** Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

4.2M

72

Snapchat Is Finally Getting a Desktop Version

## What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

### **Advantage of Exception Handling**

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in [Java](https://www.javatpoint.com/java-tutorial).

## Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



### **Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



## Difference between Checked and Unchecked Exceptions

### **1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

## Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| Try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| Catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| Finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. |
| Throw | The "throw" keyword is used to throw an exception. |
| Throws | The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

## Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

**JavaExceptionExample.java**

1. **public** **class** JavaExceptionExample{
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //code that may raise exception
5. **int** data=100/0;
6. }**catch**(ArithmeticException e){
7. System.out.println(e);
8. }
9. //rest code of the program
10. System.out.println("rest of the code...");
11. }
12. }

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

**Output:**

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

## Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

### **1) A scenario where ArithmeticException occurs**

If we divide any number by zero, there occurs an ArithmeticException.

1. **int** a=50/0;//ArithmeticException

### **2) A scenario where NullPointerException occurs**

If we have a null value in any [variable](https://www.javatpoint.com/java-variables), performing any operation on the variable throws a NullPointerException.

1. String s=**null**;
2. System.out.println(s.length());//NullPointerException

### **3) A scenario where NumberFormatException occurs**

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a [string](https://www.javatpoint.com/java-string) variable that has characters; converting this variable into digit will cause NumberFormatException.

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

### **4) A scenario where ArrayIndexOutOfBoundsException occurs**

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

1. **int** a[]=**new** **int**[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

## Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

### **Syntax of Java try-catch**

**try**{

//code that may throw an exception

1

2

 3

}**catch**(Exception\_class\_Name ref){}

4

### **Syntax of try-finally block**

1. **try**{
2. //code that may throw an exception
3. }**finally**{}

## Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

Play Video[[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack" \t "_blank)

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

## Internal Working of Java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

## Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

### **Example 1**

**TryCatchExample1.java**

1. **public** **class** TryCatchExample1 {
3. **public** **static** **void** main(String[] args) {
5. **int** data=50/0; //may throw exception
7. System.out.println("rest of the code");
9. }
11. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample1" \t "_blank)**

**Output:**

Exception in thread "main" java.lang.ArithmeticException: / by zero

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

## Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

### **Example 2**

**TryCatchExample2.java**

1. **public** **class** TryCatchExample2 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
7. }
8. //handling the exception
9. **catch**(ArithmeticException e)
10. {
11. System.out.println(e);
12. }
13. System.out.println("rest of the code");
14. }
16. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample2" \t "_blank)**

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

## Try block in Java

As mentioned in the beginning, try block contains set of statements where an exception can occur. A try block is always followed by a catch block or finally block, if exception occurs, the rest of the statements in the try block are skipped and the flow immediately jumps to the corresponding catch block.

**Note:** A try block must be followed by catch blocks or finally block or both.

### Syntax of try block with catch block

try{

//statements that may cause an exception

}catch(Exception e){

//statements that will execute when exception occurs

}

### Syntax of try block with finally block

try{

//statements that may cause an exception

}finally{

//statements that execute whether the exception occurs or not

}

### Syntax of try-catch-finally in Java

try{

//statements that may cause an exception

}catch(Exception e){

//statements that will execute if exception occurs

}finally{

//statements that execute whether the exception occurs or not

}

**Note:** It is upto the programmer to choose which statements needs to be placed inside try block. If programmer thinks that certain statements in a program can throw a exception, such statements can be enclosed inside try block and potential exceptions can be handled in catch blocks.

## Catch block in Java

A catch block is where you handle the exceptions, this block must immediately placed after a try block. **A single try block can have several catch blocks associated with it**. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

### Syntax of try catch in java

try

{

//statements that may cause an exception

}

catch (exception(type) e(object))‏

{

//error handling code

}

## Example: try catch in Java

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. As discussed earlier, a single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below).

The **generic exception handler** can handle all the exceptions but you should place is at the end, if you place it at the before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather then a generic message.

class Example1 {

public static void main(String args[]) {

int num1, num2;

try {

  /\* We suspect that this block of statement can throw

\* exception so we handled it by placing these statements

\* inside try and handled the exception in catch block

\*/

num1 = 0;

num2 = 62 / num1;

System.out.println(num2);

System.out.println("Hey I'm at the end of try block");

}

  catch (ArithmeticException e) {

/\* This block will only execute if any Arithmetic exception

\* occurs in try block

\*/

System.out.println("You should not divide a number by zero");

}

catch (Exception e) {

/\* This is a generic Exception handler which means it can handle

\* all the exceptions. This will execute if the exception is not

\* handled by previous catch blocks.

\*/

System.out.println("Exception occurred");

}

System.out.println("I'm out of try-catch block in Java.");

}

}

Output:

You should not divide a number by zero

I'm out of try-catch block in Java.

## Multiple catch blocks in Java

The example we seen above is having multiple catch blocks, let’s see few rules about multiple catch blocks with the help of examples. To read this in detail, see [**catching multiple exceptions in java**](https://beginnersbook.com/2013/05/catch-multiple-exceptions/).

1. As I mentioned above, a single try block can have any number of catch blocks.

2. A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them. To see the examples of NullPointerException and ArrayIndexOutOfBoundsException, refer this article: [**Exception Handling example programs**](https://beginnersbook.com/2013/04/exception-handling-examples/).

catch(Exception e){

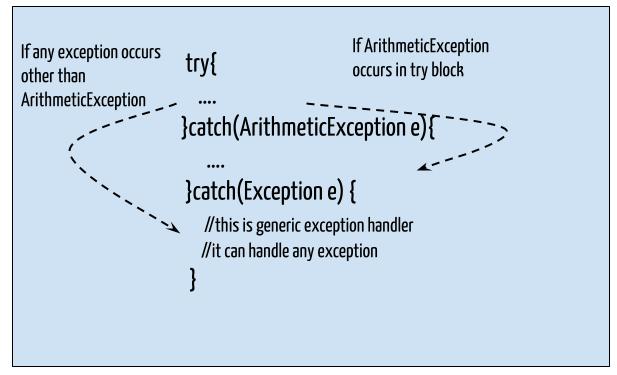
  //This catch block catches all the exceptions

}

If you are wondering why we need other catch handlers when we have a generic that can handle all. This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place is at the end of all the specific exception catch blocks

3. If **no exception** occurs in try block then the **catch blocks are completely ignored**.

4. Corresponding catch blocks execute for that specific type of exception:  
catch(ArithmeticException e) is a catch block that can handle ArithmeticException  
catch(NullPointerException e) is a catch block that can handle NullPointerException

5. You can also throw exception, which is an advanced topic and I have covered it in separate tutorials: [**user defined exception**](https://beginnersbook.com/2013/04/user-defined-exception-in-java/), [**throws keyword**](https://beginnersbook.com/2013/12/throws-keyword-example-in-java/), [**throw vs throws**](https://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/).  


### Example of Multiple catch blocks

class Example2{

public static void main(String args[]){

try{

int a[]=new int[7];

a[4]=30/0;

System.out.println("First print statement in try block");

}

catch(ArithmeticException e){

System.out.println("Warning: ArithmeticException");

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("Warning: ArrayIndexOutOfBoundsException");

}

catch(Exception e){

System.out.println("Warning: Some Other exception");

}

System.out.println("Out of try-catch block...");

}

}

Output:

Warning: ArithmeticException

Out of try-catch block...

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block ( catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it can handle all exceptions. This catch block should be placed at the last to avoid such situations.

## Finally block

# I have covered this in a separate tutorial here: [**java finally block**](https://beginnersbook.com/2013/04/java-finally-block/). For now you just need to know that **this block executes whether an exception occurs or not**. You should place those statements in finally blocks, that must execute whether exception occ

# How to Catch multiple exceptions

In the [**previous tutorial**](https://beginnersbook.com/2013/04/try-catch-in-java/), I have covered how to handle exceptions using try-catch blocks. In this guide, we will see how to handle multiple exceptions and how to write them in a correct order so that user gets a meaningful message for each type of exception.

## Catching multiple exceptions

Lets take an example to understand how to handle multiple exceptions.

class Example{

public static void main(String args[]){

try{

int arr[]=new int[7];

arr[4]=30/0;

System.out.println("Last Statement of try block");

}

catch(ArithmeticException e){

System.out.println("You should not divide a number by zero");

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("Accessing array elements outside of the limit");

}

catch(Exception e){

System.out.println("Some Other Exception");

}

System.out.println("Out of the try-catch block");

}

}

**Output:**

You should not divide a number by zero

Out of the try-catch block

In the above example, the first catch block got executed because the code we have written in try block throws ArithmeticException (because we divided the number by zero).

**Now lets change the code a little bit and see the change in output:**

class Example{

public static void main(String args[]){

try{

int arr[]=new int[7];

arr[10]=10/5;

System.out.println("Last Statement of try block");

}

catch(ArithmeticException e){

System.out.println("You should not divide a number by zero");

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("Accessing array elements outside of the limit");

}

catch(Exception e){

System.out.println("Some Other Exception");

}

System.out.println("Out of the try-catch block");

}

}

Output:

Accessing array elements outside of the limit

Out of the try-catch block

In this case, the second catch block got executed because the code throws ArrayIndexOutOfBoundsException. We are trying to access the 11th element of array in above program but the array size is only 7.

**What did we observe from the above two examples?**  
1. It is clear that when an exception occurs, the specific catch block (that declares that exception) executes. This is why in first example first block executed and in second example second catch.  
2. Although I have not shown you above, but if an exception occurs in above code which is not Arithmetic and ArrayIndexOutOfBounds then the last generic catch handler would execute.

**Lets change the code again and see the output:**

class Example{

public static void main(String args[]){

try{

int arr[]=new int[7];

arr[10]=10/5;

System.out.println("Last Statement of try block");

}

catch(Exception e){

System.out.println("Some Other Exception");

}

catch(ArithmeticException e){

System.out.println("You should not divide a number by zero");

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("Accessing array elements outside of the limit");

}

System.out.println("Out of the try-catch block");

}

}

**Output:**

Compile time error: Exception in thread "main" java.lang.Error:

Unresolved compilation problems: Unreachable catch block for ArithmeticException.

It is already handled by the catch block for Exception Unreachable catch block

for ArrayIndexOutOfBoundsException. It is already handled by the catch block for

Exception at Example.main(Example1.java:11)

**Why we got this error?**  
This is because we placed the generic exception catch block at the first place which means that none of the catch blocks placed after this block is reachable. You should always place this block at the end of all other specific exception catch blocks.

# Nested try catch block in Java – Exception handling

When a [**try catch block**](https://beginnersbook.com/2013/04/try-catch-in-java/) is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular [**exception**](https://beginnersbook.com/2013/04/java-exception-handling/), then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.

If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don’t handle exception.

Lets see the syntax first then we will discuss this with an example.

## Syntax of Nested try Catch

....

//Main try block

try {

statement 1;

statement 2;

//try-catch block inside another try block

try {

statement 3;

statement 4;

//try-catch block inside nested try block

try {

statement 5;

statement 6;

}

catch(Exception e2) {

//Exception Message

}

}

catch(Exception e1) {

//Exception Message

}

}

//Catch of Main(parent) try block

catch(Exception e3) {

//Exception Message

}

....

## Nested Try Catch Example

Here we have deep (two level) nesting which means we have a try-catch block inside a nested try block. To make you understand better I have given the names to each try block in comments like try-block2, try-block3 etc.

This is how the structure is: try-block3 is inside try-block2 and try-block2 is inside main try-block, you can say that the main try-block is a grand parent of the try-block3. Refer the explanation which is given at the end of this code.

class NestingDemo{

public static void main(String args[]){

//main try-block

try{

//try-block2

try{

//try-block3

try{

int arr[]= {1,2,3,4};

/\* I'm trying to display the value of

\* an element which doesn't exist. The

\* code should throw an exception

\*/

System.out.println(arr[10]);

}catch(ArithmeticException e){

System.out.print("Arithmetic Exception");

System.out.println(" handled in try-block3");

}

}

catch(ArithmeticException e){

System.out.print("Arithmetic Exception");

System.out.println(" handled in try-block2");

}

}

catch(ArithmeticException e3){

System.out.print("Arithmetic Exception");

System.out.println(" handled in main try-block");

}

catch(ArrayIndexOutOfBoundsException e4){

System.out.print("ArrayIndexOutOfBoundsException");

System.out.println(" handled in main try-block");

}

catch(Exception e5){

System.out.print("Exception");

System.out.println(" handled in main try-block");

}

}

}

Output:

ArrayIndexOutOfBoundsException handled in main try-block

As you can see that the ArrayIndexOutOfBoundsException occurred in the grand child try-block3. Since try-block3 is not handling this exception, the control then gets transferred to the parent try-block2 and looked for the catch handlers in try-block2. Since the try-block2 is also not handling that exception, the control gets transferred to the main (grand parent) try-block where it found the appropriate catch block for exception. This is how the the nesting structure works.

## Example 2: Nested try block

class Nest{

public static void main(String args[]){

//Parent try block

try{

//Child try block1

try{

System.out.println("Inside block1");

int b =45/0;

System.out.println(b);

}

catch(ArithmeticException e1){

System.out.println("Exception: e1");

}

//Child try block2

try{

System.out.println("Inside block2");

int b =45/0;

System.out.println(b);

}

catch(ArrayIndexOutOfBoundsException e2){

System.out.println("Exception: e2");

}

System.out.println("Just other statement");

}

catch(ArithmeticException e3){

System.out.println("Arithmetic Exception");

System.out.println("Inside parent try catch block");

}

catch(ArrayIndexOutOfBoundsException e4){

System.out.println("ArrayIndexOutOfBoundsException");

System.out.println("Inside parent try catch block");

}

catch(Exception e5){

System.out.println("Exception");

System.out.println("Inside parent try catch block");

}

System.out.println("Next statement..");

}

}

**Output:**

Inside block1

Exception: e1

Inside block2

Arithmetic Exception

Inside parent try catch block

Next statement..

This is another example that shows how the nested try block works. You can see that there are two try-catch block inside main try block’s body. I’ve marked them as block 1 and block 2 in above example.  
**Block1:**I have divided an integer by zero and it caused an ArithmeticException, since the catch of block1 is handling ArithmeticException "Exception: e1" displayed.

**Block2:**In block2, ArithmeticException occurred but block 2 catch is only handling ArrayIndexOutOfBoundsException so in this case control jump to the Main try-catch(parent) body and checks for the ArithmeticException catch handler in parent catch blocks. Since catch of parent try block is handling this exception using generic Exception handler that handles all exceptions, the message “Inside parent try catch block” displayed as output.

**Parent try Catch block:** No exception occurred here so the “Next statement..” displayed.

The important point to note here is that whenever the child catch blocks are not handling any exception, the jumps to the parent catch blocks, if the exception is not handled there as well then the program will terminate abruptly showing system generated message.

# Java Finally block – Exception handling:

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

## Syntax of Finally block

try {

//Statements that may cause an exception

}

catch {

//Handling exception

}

finally {

//Statements to be executed

}

## A Simple Example of finally block

Here you can see that the exception occurred in try block which has been handled in catch block, after that finally block got executed.

class Example

{

public static void main(String args[]) {

try{

int num=121/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("Number should not be divided by zero");

}

/\* Finally block will always execute

\* even if there is no exception in try block

\*/

finally{

System.out.println("This is finally block");

}

System.out.println("Out of try-catch-finally");

}

}

**Output:**

Number should not be divided by zero

This is finally block

Out of try-catch-finally

## Few Important points regarding finally block

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always

3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.

4. An exception in the finally block, behaves exactly like any other exception.

5. The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.  
Lets see an example to see how finally works when return statement is present in try block:

### Another example of finally block and return statement

You can see that even though we have return statement in the method, the finally block still runs.

class JavaFinally

{

public static void main(String args[])

{

System.out.println(JavaFinally.myMethod());

}

public static int myMethod()

{

try {

return 112;

}

finally {

System.out.println("This is Finally block");

System.out.println("Finally block ran even after return statement");

}

}

}

**Output of above program:**

This is Finally block

Finally block ran even after return statement

112

.

## Cases when the finally block doesn’t execute

The circumstances that prevent execution of the code in a finally block are:  
– The death of a Thread  
– Using of the System. exit() method.  
– Due to an exception arising in the finally block.

## Finally and Close()

**close()** statement is used to close all the open streams in a program. Its a good practice to use close() inside finally block. Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

For example:

....

try{

OutputStream osf = new FileOutputStream( "filename" );

OutputStream osb = new BufferedOutputStream(opf);

ObjectOutput op = new ObjectOutputStream(osb);

try{

output.writeObject(writableObject);

}

finally{

op.close();

}

}

catch(IOException e1){

System.out.println(e1);

}

...

## Finally block without catch

A try-finally block is possible without catch block. Which means a try block can be used with finally without having a catch block.

...

InputStream input = null;

try {

input = new FileInputStream("inputfile.txt");

}

finally {

if (input != null) {

try {

in.close();

}catch (IOException exp) {

System.out.println(exp);

}

}

}

...

## Finally block and System.exit()

**System.exit()** statement behaves differently than**return statement**. Unlike return statement whenever System.exit() gets called in try block then **Finally block** doesn’t execute. Here is a code snippet that demonstrate the same:

....

try {

//try block

System.out.println("Inside try block");

System.exit(0)

}

catch (Exception exp) {

System.out.println(exp);

}

finally {

System.out.println("Java finally block");

}

....

In the above example if the**System.exit(0)** gets called without any exception then finally won’t execute. However if any exception occurs while calling **System.exit(0)** then finally block will be executed.

## try-catch-finally block

* Either a try statement should be associated with a catch block or with finally.
* Since catch performs exception handling and finally performs the cleanup, the best approach is to use both of them.

**Syntax:**

try {

//statements that may cause an exception

}

catch (…)‏ {

//error handling code

}

finally {

//statements to be executed

}

### Examples of Try catch finally blocks

**Example 1:** The following example demonstrate the working of finally block when no exception occurs in try block

class Example1{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/3;

System.out.println(num);

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("ArrayIndexOutOfBoundsException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

**Output:**

First statement of try block

15

finally block

Out of try-catch-finally block

**Example 2:** This example shows the working of finally block when an exception occurs in try block but is not handled in the catch block:

class Example2{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/0;

System.out.println(num);

}

catch(ArrayIndexOutOfBoundsException e){

System.out.println("ArrayIndexOutOfBoundsException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

**Output:**

First statement of try block

finally block

Exception in thread "main" java.lang.ArithmeticException: / by zero

at beginnersbook.com.Example2.main(Details.java:6)

As you can see that the system generated exception message is shown but before that the finally block successfully executed.

**Example 3**: When exception occurs in try block and handled properly in catch block

class Example3{

public static void main(String args[]){

try{

System.out.println("First statement of try block");

int num=45/0;

System.out.println(num);

}

catch(ArithmeticException e){

System.out.println("ArithmeticException");

}

finally{

System.out.println("finally block");

}

System.out.println("Out of try-catch-finally block");

}

}

**Output:**

First statement of try block

ArithmeticException

finally block

Out of try-catch-finally block