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.

## **Platform**: Any hardware or software environment in which a program runs, is known as a platforJava Example

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

**Simple.java**

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

m. Since Java has a runtime environment (JRE) and API, it is called a platform.

## 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.

## Java Platforms / Editions

There are 4 platforms or editions of Java:

#### **1) Java SE (Java Standard Edition)**

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, [String](https://www.javatpoint.com/java-string), Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

#### **2) Java EE (Java Enterprise Edition)**

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, [JPA](https://www.javatpoint.com/jpa-tutorial), etc.

#### **3) Java ME (Java Micro Edition)**

It is a micro platform that is dedicated to mobile applications.

#### **4) JavaFX**

It is used to develop rich internet applications. It uses a lightweight user interface API.

Release Versions in java:

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)
18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (to be released by March 2022)

# **Features of Java**

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

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



1. [Simple](https://www.javatpoint.com/features-of-java#Simple)
2. [Object-Oriented](https://www.javatpoint.com/features-of-java#Object-Oriented)
3. [Portable](https://www.javatpoint.com/features-of-java#Portable)
4. [Platform independent](https://www.javatpoint.com/features-of-java#Platform-independent)
5. [Secured](https://www.javatpoint.com/features-of-java#Secured)
6. [Robust](https://www.javatpoint.com/features-of-java#Robust)
7. [Architecture neutral](https://www.javatpoint.com/features-of-java#Architecture-neutral)
8. [Interpreted](https://www.javatpoint.com/features-of-java#Interpreted)
9. [High Performance](https://www.javatpoint.com/features-of-java#High-Performance)
10. [Multithreaded](https://www.javatpoint.com/features-of-java#Multithreaded)
11. [Distributed](https://www.javatpoint.com/features-of-java#Distributed)
12. [Dynamic](https://www.javatpoint.com/features-of-java#Dynamic)

### **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.

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**



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++.

# **First Java Program | Hello World Example**

1. [Software Requirements](https://www.javatpoint.com/simple-program-of-java#hellojavareq)
2. [Creating Hello Java Example](https://www.javatpoint.com/simple-program-of-java#hellojavaex)
3. [Resolving javac is not recognized](https://www.javatpoint.com/simple-program-of-java#hellojavawhatjavacnot)

In this section, we will learn how to write the simple program of Java. We can write a simple hello Java program easily after installing the JDK.

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

### **The requirement for Java Hello World Example**

For executing any Java program, the following software or application must be properly installed.

* Install the JDK if you don't have installed it, [download the JDK](http://www.oracle.com/technetwork/java/javase/downloads/index.html) and install it.
* Set path of the jdk/bin directory. [http://www.javatpoint.com/how-to-set-path-in-java](https://www.javatpoint.com/how-to-set-path-in-java)
* Create the Java program
* Compile and run the Java 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

**Compilation Flow:**

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.

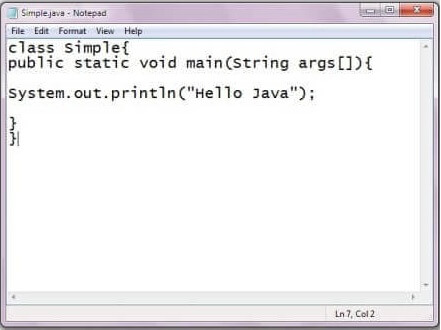


## Parameters used in First Java Program

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.

To write the simple program, you need to open notepad by **start menu -> All Programs -> Accessories -> Notepad** and write a simple program as we have shownbelow:



As displayed in the above diagram, write the simple program of Java in notepad and saved it as Simple.java. In order to compile and run the above program, you need to open the command prompt by **start menu -> All Programs -> Accessories -> command prompt**. When we have done with all the steps properly, it shows the following output:



To compile and run the above program, go to your current directory first; my current directory is c:\new. Write here:

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

## In how many ways we can write a Java program?

There are many ways to write a Java program. The modifications that can be done in a Java program are given below:

**1) By changing the sequence of the modifiers, method prototype is not changed in Java.**

Let's see the simple code of the main method.

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

**2) The subscript notation in the Java array can be used after type, before the variable or after the variable.**

Let's see the different codes to write the main method.

1. **public** **static** **void** main(String[] args)
2. **public** **static** **void** main(String []args)
3. **public** **static** **void** main(String args[])

**3) You can provide var-args support to the main() method by passing 3 ellipses (dots)**

Let's see the simple code of using var-args in the main() method. We will learn about var-args later in the Java New Features chapter.

1. **public** **static** **void** main(String... args)

**4) Having a semicolon at the end of class is optional in Java.**

Let's see the simple code.

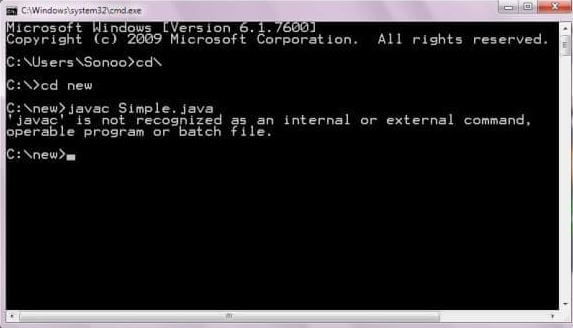
1. **class** A{
2. **static** **public** **void** main(String... args){
3. System.out.println("hello java4");
4. }
5. };

## Valid Java main() method signature

1. **public** **static** **void** main(String[] args)
2. **public** **static** **void** main(String []args)
3. **public** **static** **void** main(String args[])
4. **public** **static** **void** main(String... args)
5. **static** **public** **void** main(String[] args)
6. **public** **static** **final** **void** main(String[] args)
7. **final** **public** **static** **void** main(String[] args)
8. **final** **strictfp** **public** **static** **void** main(String[] args)

### **Resolving an error "javac is not recognized as an internal or external command"?**

If there occurs a problem like displayed in the below figure, you need to set a path. Since DOS doesn't recognize javac and java as internal or external command. To overcome this problem, we need to set a path. The path is not required in a case where you save your program inside the JDK/bin directory. However, it is an excellent approach to set the path. Click here for [How to set path in java](https://www.javatpoint.com/how-to-set-path-in-java).



# **Internal Details of Hello Java Program**

## 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.

24.3M

457

Prime Ministers of India | List of Prime Minister of India (1947-2020)

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

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

### **Q) Can you save a Java source file by another name than the class name?**

Yes, if the class is not public. It is explained in the figure given below:

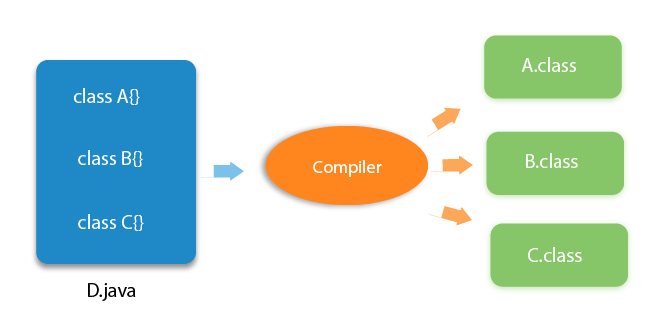


|  |  |
| --- | --- |
| **To compile:** | javac Hard.java |
| **To execute:** | java Simple |

Observe that, we have compiled the code with file name but running the program with class name. Therefore, we can save a Java program other than class name.

### **Q) Can you have multiple classes in a java source file?**

Yes, like the figure given below illustrates:



# **How to set path in Java**

1. [How to set the path of JDK in Windows OS](https://www.javatpoint.com/how-to-set-path-in-java)
   1. [Setting Temporary Path of JDK](https://www.javatpoint.com/how-to-set-path-in-java#pathtemporary)
   2. [Setting Permanent Path of JDK](https://www.javatpoint.com/how-to-set-path-in-java#pathpermanent)
2. [How to set the path of JDK in Linux OS](https://www.javatpoint.com/how-to-set-path-in-java#pathlinux)

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

35.9M

597

Hello Java Program for Beginners

1. Temporary
2. Permanent

## 1) How to set the Temporary Path of JDK in Windows

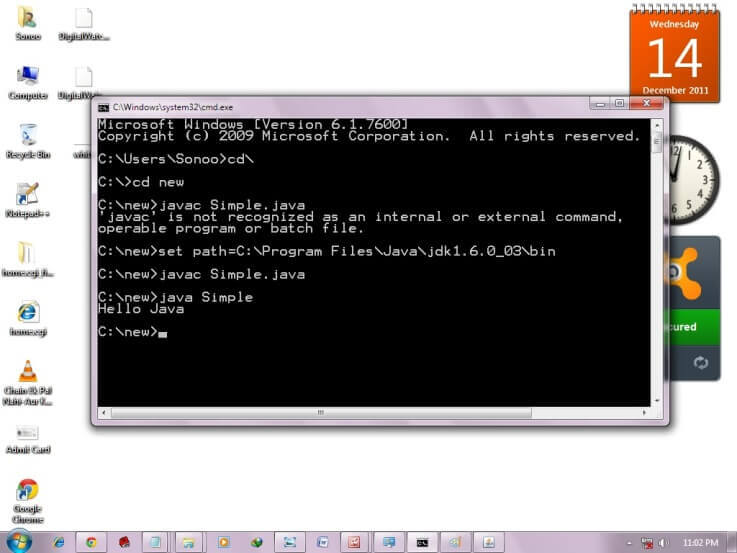
To set the temporary path of JDK, you need to follow the following steps:

* Open the command prompt
* Copy the path of the JDK/bin directory
* Write in command prompt: set path=copied\_path

### **For Example:**

set path=C:\Program Files\Java\jdk1.6.0\_23\bin

Let's see it in the figure given below:



## 2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

* Go to MyComputer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

### **For Example:**

|  |
| --- |
| **1) Go to MyComputer properties** |
| how to set path in java |
| **2) Click on the advanced tab** |
| how to set path in java |
| **3) Click on environment variables** |
| how to set path in java |
| **4) Click on the new tab of user variables** |
| how to set path in java |
| **5) Write the path in the variable name** |
| how to set path in java |
| **6) Copy the path of bin folder** |
| how to set path in java |
| **7) Paste path of bin folder in the variable value** |
| how to set path in java |
| **8) Click on ok button** |
| how to set path in java |
| **9) Click on ok button** |
| how to set path in java |

Now your permanent path is set. You can now execute any program of java from any drive.

### **Setting Java Path in Linux OS**

Setting path in Linux OS is the same as setting the path in the Windows OS. But, here we use the export tool rather than set. Let's see how to set path in Linux OS:

export PATH=$PATH:/home/jdk1.6.01/bin/

Here, we have installed the JDK in the home directory under Root (/home).

Java Coding Standards:

## 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 **drawDisplay()** { //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. | ClassEmployee { // 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;** classEmployee { //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. | classEmployee { //constant static final int  **MIN\_AGE** = 18; //code snippet } |

Control Statements in java:

* [Simple if statement](https://www.edureka.co/blog/control-statements-in-java/#Simpleifstatement)
* [if-else statement](https://www.edureka.co/blog/control-statements-in-java/#if-elsestatement)
* [Nested if statement](https://www.edureka.co/blog/control-statements-in-java/#Nestedifstatement)
* [Switch statement](https://www.edureka.co/blog/control-statements-in-java/#Switchstatement)
* [Looping statements](https://www.edureka.co/blog/control-statements-in-java/#Loopingstatemenets)
* [While](https://www.edureka.co/blog/control-statements-in-java/#While)
* [Do-while](https://www.edureka.co/blog/control-statements-in-java/#Do-while)
* [For](https://www.edureka.co/blog/control-statements-in-java/#For)
* [For-Each](https://www.edureka.co/blog/control-statements-in-java/#For-Each)
* [Branching statements](https://www.edureka.co/blog/control-statements-in-java/#Branchingstatements)
* [Break](https://www.edureka.co/blog/control-statements-in-java/#Break)
* [Continue](https://www.edureka.co/blog/control-statements-in-java/#Continue)
* **Simple if statement**
* The if statement determines whether a code should be executed based on the specified condition.  
  **Syntax:**

|  |  |
| --- | --- |
| 1  2  3  4 | if (condition) {  Statement 1; //executed if condition is true  }  Statement 2; //executed irrespective of the condition |

**If.**.**else statement**

In this statement, if the condition specified is true, the if block is executed. Otherwise, the else block is executed.  
**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Main  {  public static void main(String args[])  {  int a = 15;  if (a > 20)  System.out.println("a is greater than 10");  else  System.out.println("a is less than 10");  System.out.println("Hello World!");  }  }  } |

**Output:**  
a is less than 10  
Hello World!

**Nested if statement**

An if present inside an if block is known as a nested if block. It is similar to an if..else statement, except they are defined inside another if..else statement.  
**Syntax:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | if (condition1) {  Statement 1; //executed if first condition is true  if (condition2) {  Statement 2; //executed if second condition is true  }  else {  Statement 3; //executed if second condition is false  }  } |

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Main  {  public static void main(String args[])  {  int s = 18;  if (s > 10)  {  if (s%2==0)  System.out.println("s is an even number and greater than 10!");  else  System.out.println("s is a odd number and greater than 10!");  }  else  {  System.out.println("s is less than 10");  }  System.out.println("Hello World!");  }  } |

**Output:**  
s is an even number and greater than 10!  
Hello World!

## ****Switch statement****

A switch statement in java is used to execute a single statement from multiple conditions. The switch statement can be used with short, byte, int, long, enum types, etc.

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class Music {  public static void main(String[] args)  {  int instrument = 4;  String musicInstrument;  // switch statement with int data type  switch (instrument) {  case 1:  musicInstrument = "Guitar";  break;  case 2:  musicInstrument = "Piano";  break;  case 3:  musicInstrument = "Drums";  break;  case 4:  musicInstrument = "Flute";  break;  case 5:  musicInstrument = "Ukelele";  break;  case 6:  musicInstrument = "Violin";  break;  case 7:  musicInstrument = "Trumpet";  break;  default:  musicInstrument = "Invalid";  break;  }  System.out.println(musicInstrument);  }  } |

**Output:**  
Flute

## ****Looping Statements****

Statements that execute a block of code repeatedly until a specified condition is met are known as looping statements

## ****While****

Known as the most common loop, the while loop evaluates a certain condition. If the condition is true, the code is executed. This process is continued until the specified condition turns out to be false.

**Syntax:**

|  |  |
| --- | --- |
| 1  2  3  4 | while (condition)  {  statementOne;  } |

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class whileTest  {  public static void main(String args[])  {  int i = 5;  while (i <= 15)  {  System.out.println(i);  i = i+2;  }  }  } |

**Output:**  
5  
7  
9  
11  
13  
15

## ****Do.****.while

The do-while loop is similar to the while loop, the only difference being that the condition in the do-while loop is evaluated after the execution of the loop body.

**Syntax:**

|  |  |
| --- | --- |
| 1  2  3 | do{  //code to be executed  }while(condition); |

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Main  {  public static void main(String args[])  {  int i = 20;  do  {  System.out.println(i);  i = i+1;  } while (i <= 20);  }  } |

**Output:**  
20

**For**

The for loop in java is used to iterate and evaluate a code multiple times. When the number of iterations is known by the user, it is recommended to use the for loop.

**Syntax:**

|  |  |
| --- | --- |
| 1  2  3  4 | for (initialization; condition; increment/decrement)  {  statement;  } |

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class forLoop  {  public static void main(String args[])  {  for (int i = 1; i <= 10; i++)  System.out.println(i);  }  } |

**Output:**  
5  
6  
7  
8  
9  
10

**For-Each**

The traversal of elements in an array can be done by the for-each loop. The elements present in the array are returned one by one. It must be noted that the user does not have to increment the value in the for-each loop.

**Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class foreachLoop{  public static void main(String args[]){  int s[] = {18,25,28,29,30};  for (int i : s) {  System.out.println(i);  }  }  } |

**Output:**  
18  
25  
28  
29  
30

## ****Branching Statements****

Branching statements in java are used to jump from a statement to another statement,

**Break**

The break statement in java is used to terminate a loop and break the current flow of the program.  
 **Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Test  {  public static void main(String args[])  {  for (int i = 5; i < 10; i++)  {  if (i == 8)  break;  System.out.println(i);  }  }  } |

**Output:**  
5  
6  
7

## ****Continue****

To jump to the next iteration of the loop, we make use of the continue statement.

|  |
| --- |
| public class Main  {  public static void main(String args[])  {  for (int k = 5; k < 15; k++)  {  // Odd numbers are skipped  if (k%2 != 0)  continue;  // Even numbers are printed  System.out.print(k + " ");  }  }  } |

**Output:**  
6 8 10 12 14

## What is JDK?

JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.

In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).



## What is JRE?

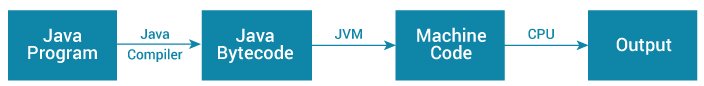
JRE (Java Runtime Environment) is a software package that provides Java class libraries, Java Virtual Machine (JVM), and other components that are required to run Java applications.

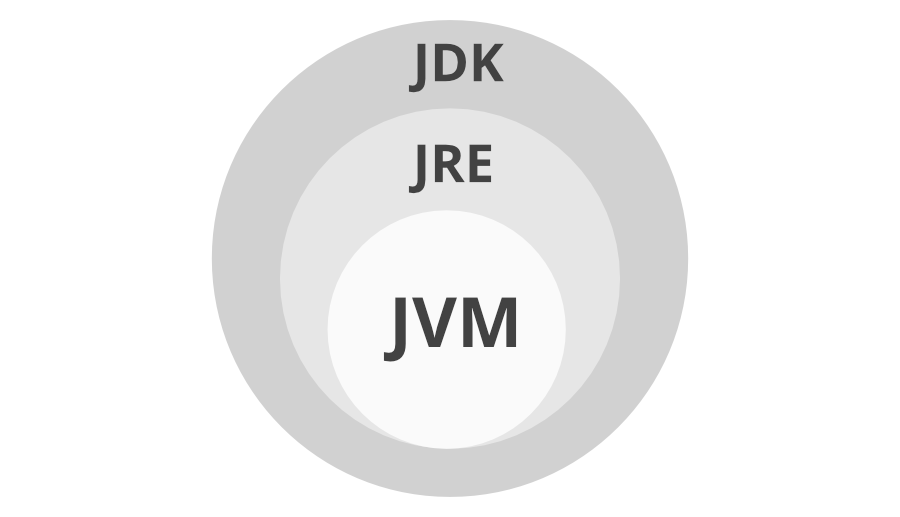
JRE is the superset of JVM.

## What is JVM?

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.

When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code .





**Java Virtual Machine (JVM)** is a engine that provides runtime environment to drive the Java Code or applications. It converts Java bytecode into machines language. JVM is a part of Java Runtime Environment (JRE).

Example:

**package** com.demo;

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

**int** id=30;

String name="abhas";

String address="hyd";

**static** **int** *salary*=6000;

**void** eat()

{

System.***out***.println("instance method");

}

**static** **void** display()

{

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

}

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

System.***out***.println(Student.*salary*);

Student.*display*();

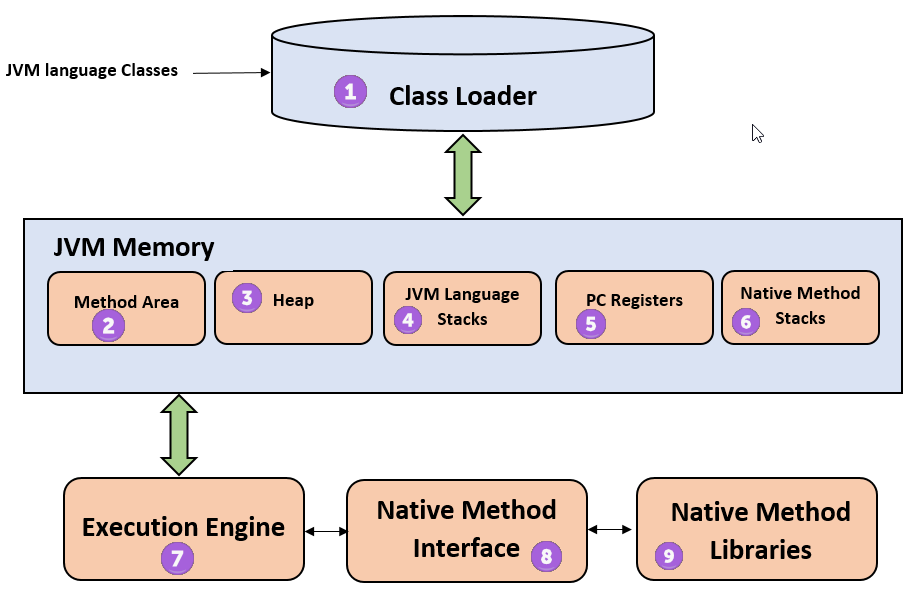
Student s=**new** Student();

s.eat();

}

}

JVM Architecture:



**1) ClassLoader**

The class loader is a subsystem used for loading class files.

### **2) Class(Method) Area**

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

### **3) Heap**

It is the runtime data area in which objects are allocated.

### **4) Stack**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

### **5) Program Counter Register**

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

### **6) Native Method Stack**

It contains all the native methods used in the application.

### **7) Execution Engine**

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

## Java Data Types:

1. **Primitive Data Types** :- which include integer, character, boolean, and float
2. **Non-primitive Data Types** :- which include classes, arrays and interfaces.

## Byte,short,char,int,long.float,double,Boolean

## Java Data Types

As the name suggests, data types specify the type of data that can be stored inside [variables in Java](https://www.programiz.com/java-programming/variables-literals).

Java is a statically-typed language. This means that all variables must be declared before they can be used.

int speed;

Here, speed is a variable, and the data type of the variable is int.

The int data type determines that the speed variable can only contain integers.

There are 8 data types predefined in Java, known as primitive data types.

**Note**: In addition to primitive data types, there are also referenced types (object type).

## 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

}

}

### 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

}

}

### 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

}

}

### 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

}

}

### 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

}

}

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

}

}

### 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

}

}

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, visit [Java single-precision and double-precision floating-point](http://stackoverflow.com/questions/801117/whats-the-difference-between-a-single-precision-and-double-precision-floating-p).

### 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

}

}

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

}

}

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. To learn more about ASCII, visit [What is ASCII Code?](https://www.ascii-code.com/).

### 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";

## Types of variables

In Java, there are three types of variables:

1. Local Variables
2. Instance Variables
3. Static Variables

### 1) Local Variables

Local Variables are a variable that are declared inside the body of a method.

### 2) Instance Variables

Instance variables are defined without the STATIC keyword .They are defined Outside a method declaration. They are Object specific and are known as instance variables.

### 3) Static Variables

Static variables are initialized only once, at the start of the program execution. These variables should be initialized first, before the initialization of any instance variables.

Example:

class Guru99 {

static int a = 1; //static variable

int data = 99; //instance variable

void method() {

int b = 90; //local variable

}

}

# Java Operators

Operators in Java can be classified into 5 types:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Unary Operators
6. Bitwise Operators

## 1. Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

a + b;

Here, the + operator is used to add two variables a and b. Similarly, there are various other arithmetic operators in Java.

|  |  |
| --- | --- |
| Operator | Operation |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulo Operation (Remainder after division) |

### Example 1: Arithmetic Operators

class Main {

public static void main(String[] args) {

// declare variables

int a = 12, b = 5;

// addition operator

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

// subtraction operator

System.out.println("a - b = " + (a - b));

// multiplication operator

System.out.println("a \* b = " + (a \* b));

// division operator

System.out.println("a / b = " + (a / b));

// modulo operator

System.out.println("a % b = " + (a % b));

}

}

**Output**

a + b = 17

a - b = 7

a \* b = 60

a / b = 2

a % b = 2

In the above example, we have used +, -, and \* operators to compute addition, subtraction, and multiplication operations.

**/ Division Operator**

Note the operation, a / b in our program. The / operator is the division operator.

If we use the division operator with two integers, then the resulting quotient will also be an integer. And, if one of the operands is a floating-point number, we will get the result will also be in floating-point.

In Java,

(9 / 2) is 4

(9.0 / 2) is 4.5

(9 / 2.0) is 4.5

(9.0 / 2.0) is 4.5

**% Modulo Operator**

The modulo operator % computes the remainder. When a = 7 is divided by b = 4, the remainder is **3**.

**Note**: The % operator is mainly used with integers.

## 2. Java Assignment Operators

Assignment operators are used in Java to assign values to variables. For example,

int age;

age = 5;

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, **5** is assigned to the variable age.

Let's see some more assignment operators available in Java.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Equivalent to |
| = | a = b; | a = b; |
| += | a += b; | a = a + b; |
| -= | a -= b; | a = a - b; |
| \*= | a \*= b; | a = a \* b; |
| /= | a /= b; | a = a / b; |
| %= | a %= b; | a = a % b; |

### Example 2: Assignment Operators

class Main {

public static void main(String[] args) {

// create variables

int a = 4;

int var;

// assign value using =

var = a;

System.out.println("var using =: " + var);

// assign value using =+

var += a;

System.out.println("var using +=: " + var);

// assign value using =\*

var \*= a;

System.out.println("var using \*=: " + var);

}

}

**Output**

var using =: 4

var using +=: 8

var using \*=: 32

## 3. Java Relational Operators

Relational operators are used to check the relationship between two operands. For example,

// check if a is less than b

a < b;

Here, < operator is the relational operator. It checks if a is less than b or not.

It returns either true or false.

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| == | Is Equal To | 3 == 5 returns **false** |
| != | Not Equal To | 3 != 5 returns **true** |
| > | Greater Than | 3 > 5 returns **false** |
| < | Less Than | 3 < 5 returns **true** |
| >= | Greater Than or Equal To | 3 >= 5 returns **false** |
| <= | Less Than or Equal To | 3 <= 5 returns **true** |

### Example 3: Relational Operators

class Main {

public static void main(String[] args) {

// create variables

int a = 7, b = 11;

// value of a and b

System.out.println("a is " + a + " and b is " + b);

// == operator

System.out.println(a == b); // false

// != operator

System.out.println(a != b); // true

// > operator

System.out.println(a > b); // false

// < operator

System.out.println(a < b); // true

// >= operator

System.out.println(a >= b); // false

// <= operator

System.out.println(a <= b); // true

}

}

**Note**: Relational operators are used in decision making and loops.

## 4. Java Logical Operators

Logical operators are used to check whether an expression is true or false. They are used in decision making.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Meaning |
| && (Logical AND) | expression1 **&&** expression2 | true only if both expression1 and expression2 are true |
| || (Logical OR) | expression1 **||** expression2 | true if either expression1 or expression2 is true |
| ! (Logical NOT) | **!**expression | true if expression is false and vice versa |

### Example 4: Logical Operators

class Main {

public static void main(String[] args) {

// && operator

System.out.println((5 > 3) && (8 > 5)); // true

System.out.println((5 > 3) && (8 < 5)); // false

// || operator

System.out.println((5 < 3) || (8 > 5)); // true

System.out.println((5 > 3) || (8 < 5)); // true

System.out.println((5 < 3) || (8 < 5)); // false

// ! operator

System.out.println(!(5 == 3)); // true

System.out.println(!(5 > 3)); // false

}

}

**Working of Program**

* (5 > 3) && (8 > 5) returns true because both (5 > 3) and (8 > 5) are true.
* (5 > 3) && (8 < 5) returns false because the expression (8 < 5) is false.
* (5 < 3) || (8 > 5) returns true because the expression (8 > 5) is true.
* (5 > 3) && (8 > 5) returns true because the expression (5 > 3) is true.
* (5 > 3) && (8 > 5) returns false because both (5 < 3) and (8 < 5) are false.
* !(5 == 3) returns true because 5 == 3 is false.
* !(5 > 3) returns false because 5 > 3 is true.

## 5. Java Unary Operators

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by **1**. That is, ++5 will return **6**.

Different types of unary operators are:

|  |  |
| --- | --- |
| Operator | Meaning |
| + | **Unary plus**: not necessary to use since numbers are positive without using it |
| - | **Unary minus**: inverts the sign of an expression |
| ++ | **Increment operator**: increments value by 1 |
| -- | **Decrement operator**: decrements value by 1 |
| ! | **Logical complement operator**: inverts the value of a boolean |

## Increment and Decrement Operators

Java also provides increment and decrement operators: ++ and -- respectively. ++ increases the value of the operand by **1**, while -- decrease it by **1**. For example,

int num = 5;

// increase num by 1

++num;

Here, the value of num gets increased to **6** from its initial value of **5**.

### Example 5: Increment and Decrement Operators

class Main {

public static void main(String[] args) {

// declare variables

int a = 12, b = 12;

int result1, result2;

// original value

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

// increment operator

result1 = ++a;

System.out.println("After increment: " + result1);

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

// decrement operator

result2 = --b;

System.out.println("After decrement: " + result2);

}

}

**Output**

Value of a: 12

After increment: 13

Value of b: 12

After decrement: 11

In the above program, we have used the ++ and -- operator as **prefixes (++a, --b)**. We can also use these operators as **postfix (a++, b++)**.

There is a slight difference when these operators are used as prefix versus when they are used as a postfix.

To learn more about these operators, visit [increment and decrement operators](https://www.programiz.com/article/increment-decrement-operator-difference-prefix-postfix).

### Java Ternary Operator

The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example,

variable = Expression ? expression1 : expression2

Here's how it works.

* If the Expression is true, expression1 is assigned to the variable.
* If the Expression is false, expression2 is assigned to the variable.

Let's see an example of a ternary operator.

class Java {

public static void main(String[] args) {

int februaryDays = 29;

String result;

// ternary operator

result = (februaryDays == 28) ? "Not a leap year" : "Leap year";

System.out.println(result);

}

}

**Output**

Leap year

## OOPs (Object-Oriented Programming System)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

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

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

* Coupling
* Cohesion
* Association
* Aggregation
* Composition



An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

### **What is an object in Java**



An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

**An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

## What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

* **Fields**
* **Methods**
* **Constructors**
* **Blocks**
* **Nested class and interface**

### **Method in Java**

In Java, a method is like a function which is used to expose the behavior of an object.

#### **Advantage of Method**

* Code Reusability
* Code Optimization

### **new keyword in Java**

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

### **Object and Class Example: Initialization through method**

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

*File: TestStudent4.java*

1. **class** Student{
2. **int** rollno;
3. String name;
4. **void** insertRecord(**int** r, String n){
5. rollno=r;
6. name=n;
7. }
8. **void** displayInformation(){
9. System.out.println(rollno+" "+name);}
10. }
11. **class** TestStudent4{
12. **public** **static** **void** main(String args[]){
13. Student s1=**new** Student();
14. Student s2=**new** Student();
15. s1.insertRecord(111,"Karan");
16. s2.insertRecord(222,"Aryan");
17. s1.displayInformation();
18. s2.displayInformation();
19. }
20. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestStudent4)

Output:

111 Karan

222 Aryan



## Java Methods

A method is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

* a method to draw the circle
* a method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

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

## Method in Java

* **Public:** The method is accessible by all classes when we use public specifier in our application.
* **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
* **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
* **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

**Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

**Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method.

**Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses.

**Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

## Types of Method

There are two types of methods in Java:

* Predefined Method
* User-defined Method

Let's see an example of the predefined method.

**Demo.java**

1. **public** **class** Demo
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. // using the max() method of Math class
6. System.out.print("The maximum number is: " + Math.max(9,7));
7. }
8. }

**Addition.java**

1. **package** com.demo;
2. **public** **class** Addition {
3. **public** **int** sum(**int** a,**int** b)
4. {
5. **int** c=a+b;
6. **return** c;
8. }
9. **public** **static** **void** main(String[] args) {
11. Addition add=**new** Addition();
13. **int** d=add.sum(10, 40);
15. System.***out***.println(d);

18. }
19. }
20. }

**Output:**

The sum of a and b is= 24

**package** com.demo;

**public** **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);

}

}

**package** com.demo;

**public** **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);

}

}

# **Constructors in Java**

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

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.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

**Note:** It is called constructor because it constructs the values at the time of object creation

### **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 Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

## Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

## Example of default constructor

|  |
| --- |
| In this example, we are creating the no-arg constructor in the Bike class.  It will be invoked at the time of object creation. |

//Java Program to create and call a default constructor

1. **class** Bike1{
2. //creating a default constructor
3. Bike1(){
4. System.out.println("Bike is created");
5. }
6. //main method
7. **public** **static** **void** main(String args[]){
8. //calling a default constructor
9. Bike1 b=**new** Bike1();
10. }
11. }

### **Q) What is the purpose of a default constructor?**

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

### **Example of default constructor that displays the default values**

1. //Let us see another example of default constructor
2. //which displays the default values
3. **class** Student3{
4. **int** id;
5. String name;
6. //method to display the value of id and name
7. **void** display(){
8. System.out.println(id+" "+name);
9. }
11. **public** **static** **void** main(String args[]){
12. //creating objects
13. Student3 s1=**new** Student3();
14. Student3 s2=**new** Student3();
15. //displaying values of the object
16. s1.display();
17. s2.display();
18. }
19. }



### **Java Parameterized Constructor**

A constructor which has a specific number of parameters is called a parameterized constructor.

### **Why use the parameterized constructor?**

The parameterized constructor is used to provide different values to distinct objects

### **Example of parameterized constructor**

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

//Java Program to demonstrate the use of the parameterized constructor.

1. **class** Student4{
2. **int** id;
3. String name;
4. //creating a parameterized constructor
5. Student4(**int** i,String n){
6. id = i;
7. name = n;
8. }
9. //method to display the values
10. **void** display(){
11. System.out.println(id+" "+name);
12. }
14. **public** **static** **void** main(String args[]){
15. //creating objects and passing values
16. Student4 s1 = **new** Student4(111,"Karan");
17. Student4 s2 = **new** Student4(222,"Aryan");
18. //calling method to display the values of object
19. s1.display();
20. s2.display();
21. }
22. }

# **Java static keyword**

The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial)

is used for memory management mainly. We can apply static keyword with [variables](https://www.javatpoint.com/java-variables)

, methods, blocks and

. The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block

## 1) Java static variable

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.

### **Example of static variable**

//Java Program to demonstrate the use of static variable

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



## 2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

### **Example of static method**

1. //Java Program to demonstrate the use of a static method.
2. **class** Student{
3. **int** rollno;
4. String name;
5. **static** String college = "ITS";
6. //static method to change the value of static variable
7. **static** **void** change(){
8. college = "BBDIT";
9. }
10. //constructor to initialize the variable
11. Student(**int** r, String n){
12. rollno = r;
13. name = n;
14. }
15. //method to display values
16. **void** display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. **public** **class** TestStaticMethod{
20. **public** **static** **void** main(String args[]){
21. Student.change();//calling change method
22. //creating objects
23. Student s1 = **new** Student(111,"Karan");
24. Student s2 = **new** Student(222,"Aryan");
25. Student s3 = **new** Student(333,"Sonoo");
26. //calling display method
27. s1.display();
28. s2.display();
29. s3.display();
30. }
31. }

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

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

### **Another example of a static method that performs a normal calculation**

1. //Java Program to get the cube of a given number using the static method
3. **class** Calculate{
4. **static** **int** cube(**int** x){
5. **return** x\*x\*x;
6. }
8. **public** **static** **void** main(String args[]){
9. **int** result=Calculate.cube(5);
10. System.out.println(result);
11. }
12. }

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

Output:125

### **Restrictions for the static method**

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.
3. **class** A{
4. **int** a=40;//non static
6. **public** **static** **void** main(String args[]){
7. System.out.println(a);
8. }
9. }

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

Output:Compile Time Error

### **Q) Why is the Java main method static?**

Ans) It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

## 3) Java static block

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

### **Example of static block**

1. **class** A2{
2. **static**{System.out.println("static block is invoked");}
3. **public** **static** **void** main(String args[]){
4. System.out.println("Hello main");
5. }
6. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A2)

Output:static block is invoked

Hello main

# **this keyword in Java**

 In Java, this is a **reference variable** that refers to the current object.



### **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.

6.4M

78

Crypto Market Plunges $130 Billion in 24 Hours

#### **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(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. **class** TestThis1{
13. **public** **static** **void** main(String args[]){
14. Student s1=**new** Student(111,"ankit",5000f);
15. Student s2=**new** Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}

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

**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" \t "_blank)**

**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" \t "_blank)**

**Output:**

111 ankit 5000.0

112 sumit 6000.0

### **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" \t "_blank)**

**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(){
3. System.out.println("hello a");
4. }
5. A(**int** x){
6. **this**();
7. System.out.println(x);
8. }
9. }
10. **class** TestThis5{
11. **public** **static** **void** main(String args[]){
12. A a=**new** A(10);
13. }}

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

**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" \t "_blank)**

**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" \t "_blank)**

**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" \t "_blank)**

**Output:**

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

# **Inheritance in Java:**

**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 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.

**class** Teacher{

String designation = "Teacher";

String collegeName = "Edureka";

**void** does(){

System.out.println("Teaching");

}

}

**public** **class** HadoopTeacher **extends** Teacher{

String mainSubject = "Spark";

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

HadoopTeacher obj = **new** HadoopTeacher();

System.out.println(obj.collegeName);

System.out.println(obj.designation);

System.out.println(obj.mainSubject);

obj.does();

}

}

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.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

[**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

# **Aggregation in Java**

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

1. **class** Employee{
2. **int** id=20;
3. String name=”saaa”;
4. Address address=new Address();//Address is a class
5. ...
6. }

### **Why use Aggregation?**

* For Code Reusability.

#### **Address.java**

**public** **class** Address {

1. String city,state,country;
3. **public** Address(String city, String state, String country) {
4. **this**.city = city;
5. **this**.state = state;
6. **this**.country = country;
7. }
9. }
10. **public** **class** Emp {
11. **int** id;
12. String name;
13. Address address;
15. **public** Emp(**int** id, String name,Address address) {
16. **this**.id = id;
17. **this**.name = name;
18. **this**.address=address;
19. }
21. **void** display(){
22. System.out.println(id+" "+name);
23. System.out.println(address.city+" "+address.state+" "+address.country);
24. }
26. **public** **static** **void** main(String[] args) {
27. Address address1=**new** Address("gzb","UP","india");
28. Address address2=**new** Address("gno","UP","india");
30. Emp e=**new** Emp(111,"varun",address1);
31. Emp e2=**new** Emp(112,"arun",address2);
33. e.display();
34. e2.display();
36. }
37. }

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

Output:111 varun

gzb UP india

112 arun

gno UP india

Type casting in java:

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

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

Type casting

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting

There are two types of type casting:

* Widening Type Casting
* Narrowing Type Casting



* Widening Type Casting

In **Widening Type Casting**, Java automatically converts one data type to another data type.

* Both data types must be compatible with each other.
* The target type must be larger than the source type.

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

**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. }

Outp[utBe B Before conversion, the value is: 7

output:

Before conversion, int value 7

After conversion, long value 7

After conversion, float value 7.0efo

### Example: Converting int to double

class Main {

public static void main(String[] args) {

// create int type variable

int num = 10;

System.out.println("The integer value: " + num);

// convert into double type

double data = num;

System.out.println("The double value: " + data);

}

}

**Output**

The integer value: 10

The double value: 10.0

re conversion, the value is: 7

### **Narrowing Type Casting**

In **Narrowing Type Casting**, we manually convert one data type into another using the parenthesis.

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

After conversion, the long value is: 7

After conversion, the float value is: 7.0

1. fore conversion, the value is: 7
2. After conversion, the long value is: 7
3. **public** **class** NarrowingTypeCastingExample
4. {
5. **public** **static** **void** main(String args[])
6. {
7. **double** d = 166.66;
8. //converting double data type into long data type
9. **long** l = (**long**)d;
10. //converting long data type into int data type
11. **int** i = (**int**)l;
12. System.***out***.println("Before conversion: "+d);
13. //fractional part lost
14. System.***out***.println("After conversion into long type: "+l);
15. //fractional part lost
16. System.***out***.println("After conversion into int type: "+i);
17. }
18. }
19. Before conversion, the value is: 7
20. After conversion, the long value is: 7
21. After conversion, the float value is: 7.0

# Java Polymorphism

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();

}

}

**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();

}

}

**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('#');

}

}

**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 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();

}

}

**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.

# **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:

Skip Ad

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(){
3. System.out.println("eating...");
4. }
5. }
6. **class** Dog **extends** Animal{
7. **void** eat(){System.out.println("eating bread...");}
8. **void** bark(){System.out.println("barking...");}
9. **void** work(){
10. **super**.eat();
11. bark();
12. }
13. }
14. **class** TestSuper2{
15. **public** **static** **void** main(String args[]){
16. Dog d=**new** Dog();
17. d.work();
18. }}

**[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(){
3. System.out.println("animal is created");
4. }
5. }
6. **class** Dog **extends** Animal{
7. Dog(){
8. **super**();
9. System.out.println("dog is created");
10. }
11. }
12. **class** TestSuper3{
13. **public** **static** **void** main(String args[]){
14. Dog d=**new** Dog();
15. }}

**[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 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);

}

}

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();

}

}

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();

}

}

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 {

^

### **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.

### **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 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:

1. **abstract** **class** Bike{
2. abstract  **void** run();

}

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

*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. Shape 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(){
4. System.out.println("bike is created");
5. }
6. **abstract** **void** run();
7. **void** changeGear(){
8. System.out.println("gear changed");
9. }
10. }
11. //Creating a Child class which inherits Abstract class
12. **class** Honda **extends** Bike{
13. **void** run(){
14. System.out.println("running safely..");
15. }
16. }
17. //Creating a Test class which calls abstract and non-abstract methods
18. **class** TestAbstraction2{
19. **public** **static** **void** main(String args[]){
20. Bike obj = **new** Honda();
21. obj.run();
22. obj.changeGear();
23. }
24. }

[**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.**

# Java Interface:

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);

}

}

**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");

}

}

**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

}

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** show();
6. }
7. **class** A7 **implements** Printable,Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **void** show(){System.out.println("Welcome");}
11. **public** **static** **void** main(String args[]){
12. A7 obj = **new** A7();
13. obj.print();
14. obj.show();
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A7)

Output:Hello

Welcome

## Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** print();
6. }
8. **class** TestInterface3 **implements** Printable, Showable{
9. **public** **void** print(){
10. System.out.println("Hello");
11. }
12. **public** **static** **void** main(String args[]){
13. TestInterface3 obj = **new** TestInterface3();
14. obj.print();
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface3)

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

## 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,

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
| 1 | An abstract class can extend only one class or one abstract class at a time | An interface can extend any number of interfaces at a time |
| 2 | An abstract class can extend another concrete (regular) class or abstract class | An interface can only extend another interface |
| 3 | An abstract class can have both abstract and concrete methods | An interface can have only abstract methods |
| 4 | In abstract class keyword “abstract” is mandatory to declare a method as an abstract | In an interface keyword “abstract” is optional to declare a method as an abstract |
| 5 | An abstract class can have protected and public abstract methods | An interface can have only have public abstract methods |
| 6 | An abstract class can have static, final or static final variable with any [**access specifier**](https://beginnersbook.com/2013/05/java-access-modifiers/) | interface can only have public static final (constant) variable |

|  |  |  |
| --- | --- | --- |
| 1 | An abstract class can extend only one class or one abstract class at a time | An interface can extend any number of interfaces at a time |
| 2 | An abstract class can extend another concrete (regular) class or abstract class | An interface can only extend another interface |
| 3 | An abstract class can have both abstract and concrete methods | An interface can have only abstract methods |
| 4 | In abstract class keyword “abstract” is mandatory to declare a method as an abstract | In an interface keyword “abstract” is optional to declare a method as an abstract |
| 5 | An abstract class can have protected and public abstract methods | An interface can have only have public abstract methods |
| 6 | An abstract class can have static, final or static final variable with any [**access specifier**](https://beginnersbook.com/2013/05/java-access-modifiers/) | interface can only have public static final (constant) variable |

# **Access Modifiers in Java**

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.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **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.

### **1) Private**

The private access modifier is accessible only within the class.

**Simple example of private access modifier**

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

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

### **Role of Private Constructor**

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

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

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

### **2) Default**

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

**Example of default access modifier**

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

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

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

### **3) Protected**

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

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

It provides more accessibility than the default modifer.

**Example of protected access modifier**

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

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

Output:Hello

### **4) Public**

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

**Example of public access modifier**

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

Output:Hello

# **Encapsulation in Java**

**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 **Java Bean** class is the example of a fully encapsulated class.

### **Advantage of Encapsulation in Java**

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### **Simple Example of Encapsulation in Java**

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

*File: Student.java*

1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **package** com.javatpoint;
4. **public** **class** Student{
5. //private data member
6. **private** String name;
7. //getter method for name
8. **public** String getName(){
9. **return** name;
10. }
11. //setter method for name
12. **public** **void** setName(String name){
13. **this**.name=name
14. }
15. }

*File: Test.java*

1. //A Java class to test the encapsulated class.
2. **package** com.javatpoint;
3. **class** Test{
4. **public** **static** **void** main(String[] args){
5. //creating instance of the encapsulated class
6. Student s=**new** Student();
7. //setting value in the name member
8. s.setName("vijay");
9. //getting value of the name member
10. System.out.println(s.getName());
11. }
12. }

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output:

vijay

### **Read-Only class**

1. //A Java class which has only getter methods.
2. **public** **class** Student{
3. //private data member
4. **private** String college="AKG";
5. //getter method for college
6. **public** String getCollege(){
7. **return** college;
8. }
9. }

Now, you can't change the value of the college data member which is "AKG".

1. s.setCollege("KITE");//will render compile time error

### **Write-Only class**

1. //A Java class which has only setter methods.
2. **public** **class** Student{
3. //private data member
4. **private** String college;
5. //getter method for college
6. **public** **void** setCollege(String college){
7. **this**.college=college;
8. }
9. }

Now, you can't get the value of the college, you can only change the value of college data member.

1. System.out.println(s.getCollege());//Compile Time Error, because there is no such method
2. System.out.println(s.college);//Compile Time Error, because the college data member is private.
3. //So, it can't be accessed from outside the class

### **Another Example of Encapsulation in Java**

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

*File: Account.java*

1. //A Account class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **class** Account {
4. //private data members
5. **private** **long** acc\_no;
6. **private** String name,email;
7. **private** **float** amount;
8. //public getter and setter methods
9. **public** **long** getAcc\_no() {
10. **return** acc\_no;
11. }
12. **public** **void** setAcc\_no(**long** acc\_no) {
13. **this**.acc\_no = acc\_no;
14. }
15. **public** String getName() {
16. **return** name;
17. }
18. **public** **void** setName(String name) {
19. **this**.name = name;
20. }
21. **public** String getEmail() {
22. **return** email;
23. }
24. **public** **void** setEmail(String email) {
25. **this**.email = email;
26. }
27. **public** **float** getAmount() {
28. **return** amount;
29. }
30. **public** **void** setAmount(**float** amount) {
31. **this**.amount = amount;
32. }
34. }

*File: TestAccount.java*

1. //A Java class to test the encapsulated class Account.
2. **public** **class** TestEncapsulation {
3. **public** **static** **void** main(String[] args) {
4. //creating instance of Account class
5. Account acc=**new** Account();
6. //setting values through setter methods
7. acc.setAcc\_no(7560504000L);
8. acc.setName("Sonoo Jaiswal");
9. acc.setEmail("sonoojaiswal@javatpoint.com");
10. acc.setAmount(500000f);
11. //getting values through getter methods
12. System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
13. }
14. }

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

Output:

7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0

# **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.

### **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. |

Object Cloning:

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

### **Example of clone() method (Object cloning)**

Let's see the simple example of object cloning

1. **class** Student18 **implements** Cloneable{
2. **int** rollno;
3. String name;
5. Student18(**int** rollno,String name){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. }
10. **public** Object clone()**throws** CloneNotSupportedException{
11. **return** **super**.clone();
12. }
14. **public** **static** **void** main(String args[]){
15. **try**{
16. Student18 s1=**new** Student18(101,"amit");
18. Student18 s2=(Student18)s1.clone();
20. System.out.println(s1.rollno+" "+s1.name);
21. System.out.println(s2.rollno+" "+s2.name);
23. }**catch**(CloneNotSupportedException c){}
25. }
26. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Student18)

Output:101 amit

101 amit

[**Next →**](https://www.javatpoint.com/wrapper-class-in-java)[**← Prev**](https://www.javatpoint.com/java-math)

# **Java Math class**

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.

For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

28.4M

600

## Example 1

1. **public** **class** JavaMathExample1
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **double** x = 28;
6. **double** y = 4;
8. // return the maximum of two numbers
9. System.out.println("Maximum number of x and y is: " +Math.max(x, y));
11. // return the square root of y
12. System.out.println("Square root of y is: " + Math.sqrt(y));
14. //returns 28 power of 4 i.e. 28\*28\*28\*28
15. System.out.println("Power of x and y is: " + Math.pow(x, y));
17. // return the logarithm of given value
18. System.out.println("Logarithm of x is: " + Math.log(x));
19. System.out.println("Logarithm of y is: " + Math.log(y));
21. // return the logarithm of given value when base is 10
22. System.out.println("log10 of x is: " + Math.log10(x));
23. System.out.println("log10 of y is: " + Math.log10(y));
25. // return the log of x + 1
26. System.out.println("log1p of x is: " +Math.log1p(x));
28. // return a power of 2
29. System.out.println("exp of a is: " +Math.exp(x));
31. // return (a power of 2)-1
32. System.out.println("expm1 of a is: " +Math.expm1(x));
33. }
34. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaMathExample1)

**Output:**

Maximum number of x and y is: 28.0

Square root of y is: 2.0

Power of x and y is: 614656.0

Logarithm of x is: 3.332204510175204

Logarithm of y is: 1.3862943611198906

log10 of x is: 1.4471580313422192

log10 of y is: 0.6020599913279624

log1p of x is: 3.367295829986474

exp of a is: 1.446257064291475E12

expm1 of a is: 1.446257064290475E12

## Example 2

1. **public** **class** JavaMathExample2
2. {
3. **public** **static** **void** main(String[] args)
4. {
5. **double** a = 30;
7. // converting values to radian
8. **double** b = Math.toRadians(a);
10. // return the trigonometric sine of a
11. System.out.println("Sine value of a is: " +Math.sin(a));
13. // return the trigonometric cosine value of a
14. System.out.println("Cosine value of a is: " +Math.cos(a));
16. // return the trigonometric tangent value of a
17. System.out.println("Tangent value of a is: " +Math.tan(a));
19. // return the trigonometric arc sine of a
20. System.out.println("Sine value of a is: " +Math.asin(a));
22. // return the trigonometric arc cosine value of a
23. System.out.println("Cosine value of a is: " +Math.acos(a));
25. // return the trigonometric arc tangent value of a
26. System.out.println("Tangent value of a is: " +Math.atan(a));
28. // return the hyperbolic sine of a
29. System.out.println("Sine value of a is: " +Math.sinh(a));
31. // return the hyperbolic cosine value of a
32. System.out.println("Cosine value of a is: " +Math.cosh(a));
34. // return the hyperbolic tangent value of a
35. System.out.println("Tangent value of a is: " +Math.tanh(a));
36. }
37. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=JavaMathExample2)

**Output:**

Sine value of a is: -0.9880316240928618

Cosine value of a is: 0.15425144988758405

Tangent value of a is: -6.405331196646276

Sine value of a is: NaN

Cosine value of a is: NaN

Tangent value of a is: 1.5374753309166493

Sine value of a is: 5.343237290762231E12

Cosine value of a is: 5.343237290762231E12

Tangent value of a is: 1.0

# Java Wrapper Class

The wrapper classes in Java are used to convert primitive types (int, char, float, etc) into corresponding objects.

Each of the 8 primitive types has corresponding wrapper classes.

|  |  |
| --- | --- |
| Primitive Type | Wrapper Class |
| Byte | Byte |
| Boolean | Boolean |
| Char | Character |
| Double | Double |
| Float | Float |
| Int | Integer |
| Long | Long |
| Short | Short |

## Convert Primitive Type to Wrapper Objects

We can also use the valueOf() method to convert primitive types into corresponding objects.

### Example 1: Primitive Types to Wrapper Objects

class Main {

public static void main(String[] args) {

// create primitive types

int a = 5;

double b = 5.65;

//converts into wrapper objects

Integer aObj = Integer.valueOf(a);

Double bObj = Double.valueOf(b);

if(aObj instanceof Integer) {

System.out.println("An object of Integer is created.");

}

if(bObj instanceof Double) {

System.out.println("An object of Double is created.");

}

}

}

**Output**

An object of Integer is created.

An object of Double is created.

In the above example, we have used the valueOf() method to convert the primitive types into objects.

Here, we have used the instanceof operator to check whether the generated objects are of Integer or Double type or not.

However, the Java compiler can directly convert the primitive types into corresponding objects. For example,

int a = 5;

// converts into object

Integer aObj = a;

double b = 5.6;

// converts into object

Double bObj = b;

This process is known as **auto-boxing**. To learn more, visit [Java autoboxing and unboxing](https://www.programiz.com/java-programming/autoboxing-unboxing).

**Note**: We can also convert primitive types into wrapper objects using Wrapper class constructors. But the use of constructors is discarded after Java 9.

## Wrapper Objects into Primitive Types

To convert objects into the primitive types, we can use the corresponding value methods (intValue(), doubleValue(), etc) present in each wrapper class.

### Example 2: Wrapper Objects into Primitive Types

class Main {

public static void main(String[] args) {

// creates objects of wrapper class

Integer aObj = Integer.valueOf(23);

Double bObj = Double.valueOf(5.55);

// converts into primitive types

int a = aObj.intValue();

double b = bObj.doubleValue();

System.out.println("The value of a: " + a);

System.out.println("The value of b: " + b);

}

}

**Output**

The value of a: 23

The value of b: 5.55

In the above example, we have used the intValue() and doubleValue() method to convert the Integer and Double objects into corresponding primitive types.

However, the Java compiler can automatically convert objects into corresponding primitive types. For example,

Integer aObj = Integer.valueOf(2);

// converts into int type

int a = aObj;

Double bObj = Double.valueOf(5.55);

// converts into double type

double b = bObj;

This process is known as **unboxing**. To learn more, visit [Java autoboxing and unboxing](https://www.programiz.com/java-programming/autoboxing-unboxing).

## Advantages of Wrapper Classes

* In Java, sometimes we might need to use objects instead of primitive data types. For example, while working with collections.
* // error
* ArrayList<int> list = new ArrayList<>();
* // runs perfectly

ArrayList<Integer> list = new ArrayList<>();

In such cases, wrapper classes help us to use primitive data types as objects.

* We can store the null value in wrapper objects. For example,
* // generates an error int a = null;
* // runs perfectly

Integer a = null;

Java String:

In [Java](https://www.javatpoint.com/java-tutorial), string is basically an object that represents sequence of char values. An [array](https://www.javatpoint.com/array-in-java) of characters works same as Java string. For example:

string is a class under java.lang packager

**Java String** class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

### **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. |

### **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)

**Output:**

java

strings

example

# **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**

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)

**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)

**Output:**

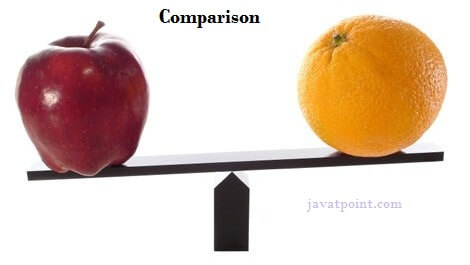
Sachin Tendulkar

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.

# **Java String compare**



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

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String in Java:

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

## 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:

28.7M

607

How to find Nth Highest Salary in SQL

* **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));
8. System.out.println(s1.equals(s3));
9. System.out.println(s1.equals(s4));
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));
7. System.out.println(s1.equalsIgnoreCase(s2));
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);
7. System.out.println(s1==s3);
8. }
9. }

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

**Output:**

true

false

## 3) String compare by compareTo() method

The above code, demonstrates the use of **==** operator used for comparing two **String** objects.

## 3) By Using compareTo() method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

* **s1 == s2** : The method returns 0.
* **s1 > s2** : The method returns a positive value.
* **s1 < s2** : The method returns a negative value.

**Teststringcomparison4.java**

1. **class** Teststringcomparison4{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3="Ratan";
6. System.out.println(s1.compareTo(s2));//0
7. System.out.println(s1.compareTo(s3));//1(because s1>s3)
8. System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Teststringcomparison4)

**Output:**

0

1

-1

# **String Concatenation in Java**

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. By + (String concatenation) operator
2. By concat() method

## 1) String Concatenation by + (String concatenation) operator

Java String concatenation operator (+) is used to add strings. For Example:

**TestStringConcatenation1.java**

1. **class** TestStringConcatenation1{
2. **public** **static** **void** main(String args[]){
3. String s="Sachin"+" Tendulkar";
4. System.out.println(s);//Sachin Tendulkar
5. }
6. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation1)

**Output:**

29.1M

592

HTML Tutorial

Sachin Tendulkar

The **Java compiler transforms** above code to this:

1. String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

In Java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and it's append method. String concatenation operator produces a new String by appending the second operand onto the end of the first operand. The String concatenation operator can concatenate not only String but primitive values also. For Example:

**TestStringConcatenation2.java**

1. **class** TestStringConcatenation2{
2. **public** **static** **void** main(String args[]){
3. String s=50+30+"Sachin"+40+40;
4. System.out.println(s);//80Sachin4040
5. }
6. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation2)

**Output:**

80Sachin4040

#### **Note: After a string literal, all the + will be treated as string concatenation operator.**

### **2) String Concatenation by concat() method**

The String concat() method concatenates the specified string to the end of current string. Syntax:

1. **public** String concat(String another)

Let's see the example of String concat() method.

**TestStringConcatenation3.java**

1. **class** TestStringConcatenation3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin ";
4. String s2="Tendulkar";
5. String s3=s1.concat(s2);
6. System.out.println(s3);//Sachin Tendulkar
7. }
8. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestStringConcatenation3)

**Output:**

Sachin Tendulkar

The above Java program, concatenates two String objects **s1** and **s2** using **concat()** method and stores the result into **s3** object.

There are some other possible ways to concatenate Strings in Java,

# **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:

* **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));
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]

# **Java String Class Methods**

The **java.lang.String** class provides a lot of built-in methods that are used to manipulate **string in Java**. By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application.

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.

36.7M

690

Difference between JDK, JRE, and JVM

**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 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. |

# **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:**

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.

# **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.

### **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.

# **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.

## 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).

Do You Know?

|  |
| --- |
| * What is the difference between checked and unchecked exceptions? * What happens behind the code int data=50/0;? * Why use multiple catch block? * Is there any possibility when the finally block is not executed? * What is exception propagation? * What is the difference between the throw and throws keyword? * What are the 4 rules for using exception handling with method overriding? |

## 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){System.out.println(e);}
7. //rest code of the program
8. System.out.println("rest of the code...");
9. }
10. }

[**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-catch block**

## 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**

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception\_class\_Name ref){}

### **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.

29.7M

574

Java Try Catch

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)

**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)

**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.

### **Example 3**

In this example, we also kept the code in a try block that will not throw an exception.

**TryCatchExample3.java**

1. **public** **class** TryCatchExample3 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
7. // if exception occurs, the remaining statement will not exceute
8. }
9. // handling the exception
10. **catch**(ArithmeticException e)
11. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
16. }
18. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample3)

**Output:**

java.lang.ArithmeticException: / by zero

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

### **Example 4**

Here, we handle the exception using the parent class exception.

**TryCatchExample4.java**

1. **public** **class** TryCatchExample4 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
7. }
8. // handling the exception by using Exception class
9. **catch**(Exception 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=TryCatchExample4)

**Output:**

java.lang.ArithmeticException: / by zero

rest of the code

### **Example 5**

Let's see an example to print a custom message on exception.

**TryCatchExample5.java**

1. **public** **class** TryCatchExample5 {
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**(Exception e)
10. {
11. // displaying the custom message
12. System.out.println("Can't divided by zero");
13. }
14. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample5)

**Output:**

Can't divided by zero

### **Example 6**

Let's see an example to resolve the exception in a catch block.

**TryCatchExample6.java**

1. **public** **class** TryCatchExample6 {
3. **public** **static** **void** main(String[] args) {
4. **int** i=50;
5. **int** j=0;
6. **int** data;
7. **try**
8. {
9. data=i/j; //may throw exception
10. }
11. // handling the exception
12. **catch**(Exception e)
13. {
14. // resolving the exception in catch block
15. System.out.println(i/(j+2));
16. }
17. }
18. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample6)

**Output:**

25

### **Example 7**

In this example, along with try block, we also enclose exception code in a catch block.

**TryCatchExample7.java**

1. **public** **class** TryCatchExample7 {
3. **public** **static** **void** main(String[] args) {
5. **try**
6. {
7. **int** data1=50/0; //may throw exception
9. }
10. // handling the exception
11. **catch**(Exception e)
12. {
13. // generating the exception in catch block
14. **int** data2=50/0; //may throw exception
16. }
17. System.out.println("rest of the code");
18. }
19. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample7)

**Output:**

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

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

### **Example 8**

In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

**TryCatchExample8.java**

1. **public** **class** TryCatchExample8 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** data=50/0; //may throw exception
8. }
9. // try to handle the ArithmeticException using ArrayIndexOutOfBoundsException
10. **catch**(ArrayIndexOutOfBoundsException e)
11. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
15. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample8)

**Output:**

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

### **Example 9**

Let's see an example to handle another unchecked exception.

**TryCatchExample9.java**

1. **public** **class** TryCatchExample9 {
3. **public** **static** **void** main(String[] args) {
4. **try**
5. {
6. **int** arr[]= {1,3,5,7};
7. System.out.println(arr[10]); //may throw exception
8. }
9. // handling the array exception
10. **catch**(ArrayIndexOutOfBoundsException e)
11. {
12. System.out.println(e);
13. }
14. System.out.println("rest of the code");
15. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample9)

**Output:**

java.lang.ArrayIndexOutOfBoundsException: 10

rest of the code

### **Example 10**

Let's see an example to handle checked exception.

**TryCatchExample10.java**

1. **import** java.io.FileNotFoundException;
2. **import** java.io.PrintWriter;
4. **public** **class** TryCatchExample10 {
6. **public** **static** **void** main(String[] args) {

9. PrintWriter pw;
10. **try** {
11. pw = **new** PrintWriter("jtp.txt"); //may throw exception
12. pw.println("saved");
13. }
14. // providing the checked exception handler
15. **catch** (FileNotFoundException e) {
17. System.out.println(e);
18. }
19. System.out.println("File saved successfully");
20. }
21. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TryCatchExample10)

**Output:**

File saved successfully

# **Java Catch Multiple Exceptions**

## Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

## Points to remember

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

### **Flowchart of Multi-catch Block**



### **Example 1**

Let's see a simple example of java multi-catch block.

**MultipleCatchBlock1.java**

1. **public** **class** MultipleCatchBlock1 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. **int** a[]=**new** **int**[5];
7. a[5]=30/0;
8. }
9. **catch**(ArithmeticException e)
10. {
11. System.out.println("Arithmetic Exception occurs");
12. }
13. **catch**(ArrayIndexOutOfBoundsException e)
14. {
15. System.out.println("ArrayIndexOutOfBounds Exception occurs");
16. }
17. **catch**(Exception e)
18. {
19. System.out.println("Parent Exception occurs");
20. }
21. System.out.println("rest of the code");
22. }
23. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock1)

**Output:**

37M

785

Exception Handling in Java - Javatpoint

Arithmetic Exception occurs

rest of the code

### **Example 2**

**MultipleCatchBlock2.java**

1. **public** **class** MultipleCatchBlock2 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. **int** a[]=**new** **int**[5];
8. System.out.println(a[10]);
9. }
10. **catch**(ArithmeticException e)
11. {
12. System.out.println("Arithmetic Exception occurs");
13. }
14. **catch**(ArrayIndexOutOfBoundsException e)
15. {
16. System.out.println("ArrayIndexOutOfBounds Exception occurs");
17. }
18. **catch**(Exception e)
19. {
20. System.out.println("Parent Exception occurs");
21. }
22. System.out.println("rest of the code");
23. }
24. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock2)

**Output:**

ArrayIndexOutOfBounds Exception occurs

rest of the code

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

**MultipleCatchBlock3.java**

1. **public** **class** MultipleCatchBlock3 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. **int** a[]=**new** **int**[5];
7. a[5]=30/0;
8. System.out.println(a[10]);
9. }
10. **catch**(ArithmeticException e)
11. {
12. System.out.println("Arithmetic Exception occurs");
13. }
14. **catch**(ArrayIndexOutOfBoundsException e)
15. {
16. System.out.println("ArrayIndexOutOfBounds Exception occurs");
17. }
18. **catch**(Exception e)
19. {
20. System.out.println("Parent Exception occurs");
21. }
22. System.out.println("rest of the code");
23. }
24. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock3)

**Output:**

Arithmetic Exception occurs

rest of the code

### **Example 4**

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class **Exception** will invoked.

**MultipleCatchBlock4.java**

1. **public** **class** MultipleCatchBlock4 {
3. **public** **static** **void** main(String[] args) {
5. **try**{
6. String s=**null**;
7. System.out.println(s.length());
8. }
9. **catch**(ArithmeticException e)
10. {
11. System.out.println("Arithmetic Exception occurs");
12. }
13. **catch**(ArrayIndexOutOfBoundsException e)
14. {
15. System.out.println("ArrayIndexOutOfBounds Exception occurs");
16. }
17. **catch**(Exception e)
18. {
19. System.out.println("Parent Exception occurs");
20. }
21. System.out.println("rest of the code");
22. }
23. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=MultipleCatchBlock4)

**Output:**

Parent Exception occurs

rest of the code

# **Java finally block**

**Java finally block** is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

### **Flowchart of finally block**



#### **Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).**

## Why use Java finally block?

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

## Usage of Java finally

Let's see the different cases where Java finally block can be used.

### **Case 1: When an exception does not occur**

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

**TestFinallyBlock.java**

1. **class** TestFinallyBlock {
2. **public** **static** **void** main(String args[]){
3. **try**{
4. //below code do not throw any exception
5. **int** data=25/5;
6. System.out.println(data);
7. }
8. //catch won't be executed
9. **catch**(NullPointerException e){
10. System.out.println(e);
11. }
12. //executed regardless of exception occurred or not
13. **finally** {
14. System.out.println("finally block is always executed");
15. }
17. System.out.println("rest of phe code...");
18. }
19. }

**Output:**



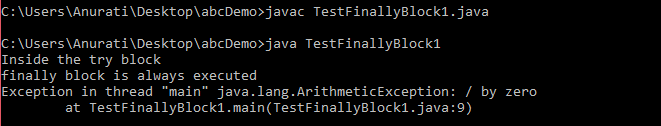
### **Case 2: When an exception occurr but not handled by the catch block**

Let's see the the fillowing example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

**TestFinallyBlock1.java**

1. **public** **class** TestFinallyBlock1{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside the try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
12. //cannot handle Arithmetic type exception
13. //can only accept Null Pointer type exception
14. **catch**(NullPointerException e){
15. System.out.println(e);
16. }
18. //executes regardless of exception occured or not
19. **finally** {
20. System.out.println("finally block is always executed");
21. }
23. System.out.println("rest of the code...");
24. }
25. }

**Output:**



### **Case 3: When an exception occurs and is handled by the catch block**

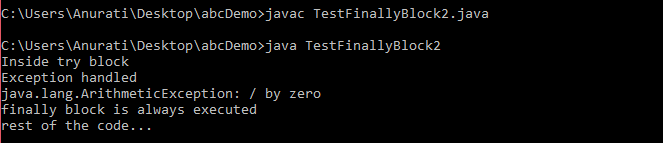
**Example:**

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

**TestFinallyBlock2.java**

1. **public** **class** TestFinallyBlock2{
2. **public** **static** **void** main(String args[]){
4. **try** {
6. System.out.println("Inside try block");
8. //below code throws divide by zero exception
9. **int** data=25/0;
10. System.out.println(data);
11. }
13. //handles the Arithmetic Exception / Divide by zero exception
14. **catch**(ArithmeticException e){
15. System.out.println("Exception handled");
16. System.out.println(e);
17. }
19. //executes regardless of exception occured or not
20. **finally** {
21. System.out.println("finally block is always executed");
22. }
24. System.out.println("rest of the code...");
25. }
26. }

**Output:**



#### **Rule: For each try block there can be zero or more catch blocks, but only one finally block.**

#### **Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).**

# **Java throw Exception**

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

## Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. **throw** **new** exception\_class("error message");

Let's see the example of throw IOException.

1. **throw** **new** IOException("sorry device error");

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

## Java throw keyword Example

### **Example 1: Throwing Unchecked Exception**

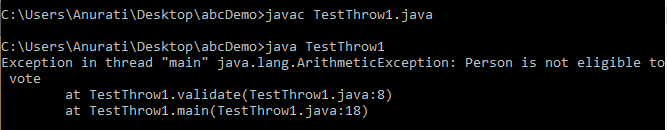
In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

**TestThrow1.java**

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

1. **public** **class** TestThrow1 {
2. //function to check if person is eligible to vote or not
3. **public** **static** **void** validate(**int** age) {
4. **if**(age<18) {
5. //throw Arithmetic exception if not eligible to vote
6. **throw** **new** ArithmeticException("Person is not eligible to vote");
7. }
8. **else** {
9. System.out.println("Person is eligible to vote!!");
10. }
11. }
12. //main method
13. **public** **static** **void** main(String args[]){
14. //calling the function
15. validate(13);
16. System.out.println("rest of the code...");
17. }
18. }

**Output:**



The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

#### **Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.**

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

### **Example 2: Throwing Checked Exception**

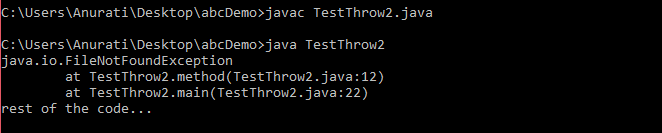
#### **Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.**

**TestThrow2.java**

1. **import** java.io.\*;
3. **public** **class** TestThrow2 {
5. //function to check if person is eligible to vote or not
6. **public** **static** **void** method() **throws** FileNotFoundException {
8. FileReader file = **new** FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
9. BufferedReader fileInput = **new** BufferedReader(file);

12. **throw** **new** FileNotFoundException();
14. }
15. //main method
16. **public** **static** **void** main(String args[]){
17. **try**
18. {
19. method();
20. }
21. **catch** (FileNotFoundException e)
22. {
23. e.printStackTrace();
24. }
25. System.out.println("rest of the code...");
26. }
27. }

**Output:**

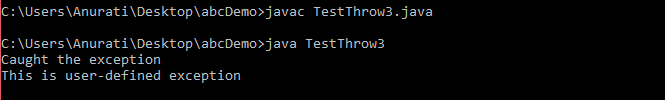


### **Example 3: Throwing User-defined Exception**

exception is everything else under the Throwable class.

**TestThrow3.java**

1. // class represents user-defined exception
2. **class** UserDefinedException **extends** Exception
3. {
4. **public** UserDefinedException(String str)
5. {
6. // Calling constructor of parent Exception
7. **super**(str);
8. }
9. }
10. // Class that uses above MyException
11. **public** **class** TestThrow3
12. {
13. **public** **static** **void** main(String args[])
14. {
15. **try**
16. {
17. // throw an object of user defined exception
18. **throw** **new** UserDefinedException("This is user-defined exception");
19. }
20. **catch** (UserDefinedException ude)
21. {
22. System.out.println("Caught the exception");
23. // Print the message from MyException object
24. System.out.println(ude.getMessage());
25. }
26. }
27. }

**Output:** 

# **Java Exception Propagation**

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

#### **Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).**

## Exception Propagation Example

**TestExceptionPropagation1.java**

1. **class** TestExceptionPropagation1{
2. **void** m(){
3. **int** data=50/0;
4. }
5. **void** n(){
6. m();
7. }
8. **void** p(){
9. **try**{
10. n();
11. }**catch**(Exception e){System.out.println("exception handled");}
12. }
13. **public** **static** **void** main(String args[]){
14. TestExceptionPropagation1 obj=**new** TestExceptionPropagation1();
15. obj.p();
16. System.out.println("normal flow...");
17. }
18. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestExceptionPropagation1" \t "_blank)**

**Output:**

exception handled

normal flow...

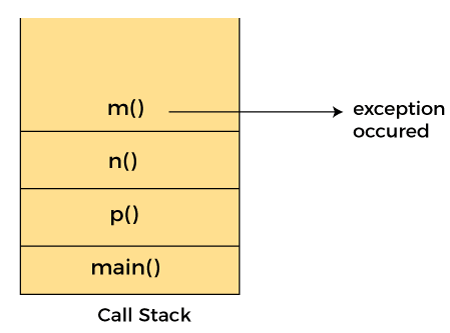
In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

34.4M

796

OOPs Concepts in Java

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



#### **Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).**

## Exception Propagation Example

**TestExceptionPropagation1.java**

1. **class** TestExceptionPropagation2{
2. **void** m(){
3. **throw** **new** java.io.IOException("device error");//checked exception
4. }
5. **void** n(){
6. m();
7. }
8. **void** p(){
9. **try**{
10. n();
11. }**catch**(Exception e){System.out.println("exception handeled");}
12. }
13. **public** **static** **void** main(String args[]){
14. TestExceptionPropagation2 obj=**new** TestExceptionPropagation2();
15. obj.p();
16. System.out.println("normal flow");
17. }
18. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestExceptionPropagation2" \t "_blank)**

**Output:**

Compile Time Error

# **Java throws keyword**

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

### **Syntax of Java throws**

1. return\_type method\_name() **throws** exception\_class\_name{
2. //method code
3. }

### **Which exception should be declared?**

**Ans:** Checked exception only, because:

* **unchecked exception:** under our control so we can correct our code.
* **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### **Advantage of Java throws keyword**

Now Checked Exception can be propagated (forwarded in call stack).

25.4M

474

Prime Ministers of India | List of Prime Minister of India (1947-2020)

It provides information to the caller of the method about the exception.

## Java throws Example

Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

**Testthrows1.java**

1. **import** java.io.IOException;
2. **class** Testthrows1{
3. **void** m()**throws** IOException{
4. **throw** **new** IOException("device error");//checked exception
5. }
6. **void** n()**throws** IOException{
7. m();
8. }
9. **void** p(){
10. **try**{
11. n();
12. }**catch**(Exception e){System.out.println("exception handled");}
13. }
14. **public** **static** **void** main(String args[]){
15. Testthrows1 obj=**new** Testthrows1();
16. obj.p();
17. System.out.println("normal flow...");
18. }
19. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Testthrows1)

**Output:**

exception handled

normal flow...

#### **Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.**

**There are two cases:**

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

### **Case 1: Handle Exception Using try-catch block**

In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

**Testthrows2.java**

1. **import** java.io.\*;
2. **class** M{
3. **void** method()**throws** IOException{
4. **throw** **new** IOException("device error");
5. }
6. }
7. **public** **class** Testthrows2{
8. **public** **static** **void** main(String args[]){
9. **try**{
10. M m=**new** M();
11. m.method();
12. }**catch**(Exception e){System.out.println("exception handled");}
14. System.out.println("normal flow...");
15. }
16. }

# **Java Custom Exception**

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

In this section, we will learn how custom exceptions are implemented and used in Java programs.

6.4M

107

Here Are Some of the Best Games of 2021

## Why use custom exceptions?

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

* To catch and provide specific treatment to a subset of existing Java exceptions.
* Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

1. **public** **class** WrongFileNameException **extends** Exception {
2. **public** WrongFileNameException(String errorMessage) {
3. **super**(errorMessage);
4. }
5. }

#### **Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.**

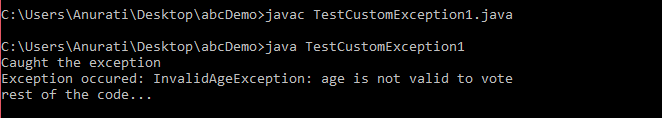
### **Example 1:**

Let's see a simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method. Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

**TestCustomException1.java**

1. // class representing custom exception
2. **class** InvalidAgeException  **extends** Exception
3. {
4. **public** InvalidAgeException (String str)
5. {
6. // calling the constructor of parent Exception
7. **super**(str);
8. }
9. }
11. // class that uses custom exception InvalidAgeException
12. **public** **class** TestCustomException1
13. {
15. // method to check the age
16. **static** **void** validate (**int** age) **throws** InvalidAgeException{
17. **if**(age < 18){
19. // throw an object of user defined exception
20. **throw** **new** InvalidAgeException("age is not valid to vote");
21. }
22. **else** {
23. System.out.println("welcome to vote");
24. }
25. }
27. // main method
28. **public** **static** **void** main(String args[])
29. {
30. **try**
31. {
32. // calling the method
33. validate(13);
34. }
35. **catch** (InvalidAgeException ex)
36. {
37. System.out.println("Caught the exception");
39. // printing the message from InvalidAgeException object
40. System.out.println("Exception occured: " + ex);
41. }
43. System.out.println("rest of the code...");
44. }
45. }

**Output:**

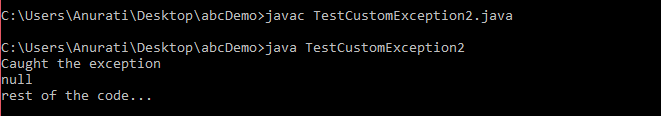


### **Example 2:**

**TestCustomException2.java**

1. // class representing custom exception
2. **class** MyCustomException **extends** Exception
3. {
5. }
7. // class that uses custom exception MyCustomException
8. **public** **class** TestCustomException2
9. {
10. // main method
11. **public** **static** **void** main(String args[])
12. {
13. **try**
14. {
15. // throw an object of user defined exception
16. **throw** **new** MyCustomException();
17. }
18. **catch** (MyCustomException ex)
19. {
20. System.out.println("Caught the exception");
21. System.out.println(ex.getMessage());
22. }
24. System.out.println("rest of the code...");
25. }
26. }

**Output:**



# Java Arrays

An array is a collection of similar types of data.

For example, if we want to store the names of 100 people then we can create an array of the string type that can store 100 names.

String[] array = new String[100];

Here, the above array cannot store more than 100 names. The number of values in a Java array is always fixed.

## How to declare an array in Java?

In Java, here is how we can declare an array.

dataType[] arrayName;

* dataType - it can be [primitive data types](https://www.programiz.com/java-programming/variables-primitive-data-types#data-types) like int, char, double, byte, etc. or [Java objects](https://www.programiz.com/java-programming/class-objects)
* arrayName - it is an [identifier](https://www.programiz.com/java-programming/keywords-identifiers#identifiers)

For example,

double[] data;

Here, data is an array that can hold values of type double.

**But, how many elements can array this hold?**

Good question! To define the number of elements that an array can hold, we have to allocate memory for the array in Java. For example,

// declare an array

double[] data;

// allocate memory

data = new double[10];

Here, the array can store **10** elements. We can also say that the **size or length** of the array is 10.

In Java, we can declare and allocate the memory of an array in one single statement. For example,

double[] data = new double[10];

## How to Initialize Arrays in Java?

In Java, we can initialize arrays during declaration. For example,

//declare and initialize and array

int[] age = {12, 4, 5, 2, 5};

Here, we have created an array named age and initialized it with the values inside the curly brackets.

Note that we have not provided the size of the array. In this case, the Java compiler automatically specifies the size by counting the number of elements in the array (i.e. 5).

In the Java array, each memory location is associated with a number. The number is known as an array index. We can also initialize arrays in Java, using the index number. For example,

// declare an array

int[] age = new int[5];

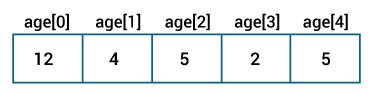
// initialize array

age[0] = 12;

age[1] = 4;

age[2] = 5;

..

Java Arrays initialization

**Note**:

* Array indices always start from 0. That is, the first element of an array is at index 0.
* If the size of an array is n, then the last element of the array will be at index n-1.

## How to Access Elements of an Array in Java?

We can access the element of an array using the index number. Here is the syntax for accessing elements of an array,

// access array elements

array[index]

Let's see an example of accessing array elements using index numbers.

### Example: Access Array Elements

class Main {

public static void main(String[] args) {

// create an array

int[] age = {12, 4, 5, 2, 5};

// access each array elements

System.out.println("Accessing Elements of Array:");

System.out.println("First Element: " + age[0]);

System.out.println("Second Element: " + age[1]);

System.out.println("Third Element: " + age[2]);

System.out.println("Fourth Element: " + age[3]);

System.out.println("Fifth Element: " + age[4]);

}

}

**Output**

Accessing Elements of Array:

First Element: 12

Second Element: 4

Third Element: 5

Fourth Element: 2

Fifth Element: 5

In the above example, notice that we are using the index number to access each element of the array.

We can use loops to access all the elements of the array at once.

## Looping Through Array Elements

In Java, we can also loop through each element of the array. For example,

### Example: Using For Loop

class Main {

public static void main(String[] args) {

// create an array

int[] age = {12, 4, 5};

// loop through the array

// using for loop

System.out.println("Using for Loop:");

for(int i = 0; i < age.length; i++) {

System.out.println(age[i]);

}

}

}

**Output**

Using for Loop:

12

4

5

In the above example, we are using the [for Loop in Java](https://www.programiz.com/java-programming/for-loop) to iterate through each element of the array. Notice the expression inside the loop,

age.length

Here, we are using the length property of the array to get the size of the array.

We can also use the [for-each loop](https://www.programiz.com/java-programming/enhanced-for-loop) to iterate through the elements of an array. For example,

### Example: Using the for-each Loop

class Main {

public static void main(String[] args) {

// create an array

int[] age = {12, 4, 5};

// loop through the array

// using for loop

System.out.println("Using for-each Loop:");

for(int a : age) {

System.out.println(a);

}

}

}

**Output**

Using for-each Loop:

12

4

5

## Example: Compute Sum and Average of Array Elements

class Main {

public static void main(String[] args) {

int[] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};

int sum = 0;

Double average;

// access all elements using for each loop

// add each element in sum

for (int number: numbers) {

sum += number;

}

// get the total number of elements

int arrayLength = numbers.length;

// calculate the average

// convert the average from int to double

average = ((double)sum / (double)arrayLength);

System.out.println("Sum = " + sum);

System.out.println("Average = " + average);

}

}

**Output**:

Sum = 36

Average = 3.6

In the above example, we have created an array of named numbers. We have used the for...each loop to access each element of the array.

Inside the loop, we are calculating the sum of each element. Notice the line,

int arrayLength = number.length;

Here, we are using the [length attribute](http://stackoverflow.com/questions/8755812/array-length-in-java) of the array to calculate the size of the array. We then calculate the average using:

average = ((double)sum / (double)arrayLength);

As you can see, we are converting the int value into double. This is called type casting in Java. To learn more about typecasting, visit [Java Type Casting](https://www.programiz.com/java-programming/typecasting).

# **Collections in Java:**

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

#### **What is Collection in Java**

A Collection represents a single unit of objects, i.e., a group.

#### **What is a framework in Java**

* It provides readymade architecture.
* It represents a set of classes and interfaces.



### **Methods of Collection interface**

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the  invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified  collection  from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that  satisfy the  specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection  except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the  collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type  of the  returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection  as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its  source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean add(E e) | It is used to append the specified element at the end of a list. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of a list. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void clear() | It is used to remove all of the elements from this list. |
|  |  |
| boolean equals(Object o) | It is used to compare the specified object with the elements of a list. |
| int hashcode() | It is used to return the hash code value for a list. |
| E get(int index) | It is used to fetch the element from the particular position of the list. |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| <T> T[] toArray(T[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| boolean contains(Object o) | It returns true if the list contains the specified element |
| boolean containsAll(Collection<?> c) | It returns true if the list contains all the specified element |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |  |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element. |  |
| boolean removeAll(Collection<?> c) | It is used to remove all the elements from the list. |  |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |  |
| void retainAll(Collection<?> c) | It is used to retain all the elements in the list that are present in the specified collection. |  |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the specified position. |  |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of specified comparator. |  |
| Spliterator<E> spliterator() | It is used to create spliterator over the elements in a list. |  |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements lies within the given range. |  |
| int size() | It is used to return the number of elements present in the list. |  |

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

### **Java List Methods**

## ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

1. **import** java.util.\*;
2. **class** TestJavaCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

# **Java ArrayList**

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here. The ArrayList maintains the insertion order internally.

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java)

.

* Java ArrayList allows random access because array works at the index basis.
* In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

### **Java ArrayList Example**

1. **import** java.util.\*;
2. **public** **class** ArrayListExample1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Mango");//Adding object in arraylist
6. list.add("Apple");
7. list.add("Banana");
8. list.add("Grapes");
9. //Printing the arraylist object
10. System.out.println(list);
11. }
12. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample1" \t "_blank)**

**Output:**

[Mango, Apple, Banana, Grapes]

### **Iterating ArrayList using Iterator**

Let's see an example to traverse ArrayList elements using the Iterator interface.

1. **import** java.util.\*;
2. **public** **class** ArrayListExample2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Mango");//Adding object in arraylist
6. list.add("Apple");
7. list.add("Banana");
8. list.add("Grapes");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();//getting the Iterator
11. **while**(itr.hasNext()){//check if iterator has the elements
12. System.out.println(itr.next());//printing the element and move to next
13. }
14. }
15. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample2" \t "_blank)**

**Output:**

Mango

Apple

Banana

Grapes

### **Iterating ArrayList using For-each loop**

Let's see an example to traverse the ArrayList elements using the for-each loop

1. **import** java.util.\*;
2. **public** **class** ArrayListExample3{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Mango");//Adding object in arraylist
6. list.add("Apple");
7. list.add("Banana");
8. list.add("Grapes");
9. //Traversing list through for-each loop
10. **for**(String fruit:list)
11. System.out.println(fruit);
13. }
14. }

**Output:**

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample3" \t "_blank)**

Mango

Apple

Banana

Grapes

### **Get and Set ArrayList**

The get() method returns the element at the specified index, whereas the set() method changes the element.

1. **import** java.util.\*;
2. **public** **class** ArrayListExample4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Mango");
6. al.add("Apple");
7. al.add("Banana");
8. al.add("Grapes");
9. //accessing the element
10. System.out.println("Returning element: "+al.get(1));//it will return the 2nd element, because index starts from 0
11. //changing the element
12. al.set(1,"Dates");
13. //Traversing list
14. **for**(String fruit:al)
15. System.out.println(fruit);
17. }
18. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample4" \t "_blank)**

**Output:**

Returning element: Apple

Mango

Dates

Banana

Grapes

### **How to Sort ArrayList**

The java.util package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort the ArrayList.

1. **import** java.util.\*;
2. **class** SortArrayList{
3. **public** **static** **void** main(String args[]){
4. //Creating a list of fruits
5. List<String> list1=**new** ArrayList<String>();
6. list1.add("Mango");
7. list1.add("Apple");
8. list1.add("Banana");
9. list1.add("Grapes");
10. //Sorting the list
11. Collections.sort(list1);
12. //Traversing list through the for-each loop
13. **for**(String fruit:list1)
14. System.out.println(fruit);
16. System.out.println("Sorting numbers...");
17. //Creating a list of numbers
18. List<Integer> list2=**new** ArrayList<Integer>();
19. list2.add(21);
20. list2.add(11);
21. list2.add(51);
22. list2.add(1);
23. //Sorting the list
24. Collections.sort(list2);
25. //Traversing list through the for-each loop
26. **for**(Integer number:list2)
27. System.out.println(number);
28. }
30. }

**Output:**

Apple

Banana

Grapes

Mango

Sorting numbers...

1

11

21

51

### **Ways to iterate the elements of the collection in Java**

There are various ways to traverse the collection elements:

1. By Iterator interface.
2. By for-each loop.
3. By ListIterator interface.
4. By for loop.
5. By forEach() method.
6. By forEachRemaining() method.

### **User-defined class objects in Java ArrayList**

Let's see an example where we are storing Student class object in an array list.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }
11. **import** java.util.\*;
12. **class** ArrayList5{
13. **public** **static** **void** main(String args[]){
14. //Creating user-defined class objects
15. Student s1=**new** Student(101,"Sonoo",23);
16. Student s2=**new** Student(102,"Ravi",21);
17. Student s2=**new** Student(103,"Hanumat",25);
18. //creating arraylist
19. ArrayList<Student> al=**new** ArrayList<Student>();
20. al.add(s1);//adding Student class object
21. al.add(s2);
22. al.add(s3);
23. //Getting Iterator
24. Iterator itr=al.iterator();
25. //traversing elements of ArrayList object
26. **while**(itr.hasNext()){
27. Student st=(Student)itr.next();
28. System.out.println(st.rollno+" "+st.name+" "+st.age);
29. }
30. }
31. }

**Output:**

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

### **Java ArrayList example to add elements**

Here, we see different ways to add an element.

1. **import** java.util.\*;
2. **class** ArrayList7{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. System.out.println("Initial list of elements: "+al);
6. //Adding elements to the end of the list
7. al.add("Ravi");
8. al.add("Vijay");
9. al.add("Ajay");
10. System.out.println("After invoking add(E e) method: "+al);
11. //Adding an element at the specific position
12. al.add(1, "Gaurav");
13. System.out.println("After invoking add(int index, E element) method: "+al);
14. ArrayList<String> al2=**new** ArrayList<String>();
15. al2.add("Sonoo");
16. al2.add("Hanumat");
17. //Adding second list elements to the first list
18. al.addAll(al2);
19. System.out.println("After invoking addAll(Collection<? extends E> c) method: "+al);
20. ArrayList<String> al3=**new** ArrayList<String>();
21. al3.add("John");
22. al3.add("Rahul");
23. //Adding second list elements to the first list at specific position
24. al.addAll(1, al3);
25. System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+al);
27. }
28. }

**Output:**

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

### **Java ArrayList example to remove elements**

Here, we see different ways to remove an element.

1. **import** java.util.\*;
2. **class** ArrayList8 {
4. **public** **static** **void** main(String [] args)
5. {
6. ArrayList<String> al=**new** ArrayList<String>();
7. al.add("Ravi");
8. al.add("Vijay");
9. al.add("Ajay");
10. al.add("Anuj");
11. al.add("Gaurav");
12. System.out.println("An initial list of elements: "+al);
13. //Removing specific element from arraylist
14. al.remove("Vijay");
15. System.out.println("After invoking remove(object) method: "+al);
16. //Removing element on the basis of specific position
17. al.remove(0);
18. System.out.println("After invoking remove(index) method: "+al);
20. //Creating another arraylist
21. ArrayList<String> al2=**new** ArrayList<String>();
22. al2.add("Ravi");
23. al2.add("Hanumat");
24. //Adding new elements to arraylist
25. al.addAll(al2);
26. System.out.println("Updated list : "+al);
27. //Removing all the new elements from arraylist
28. al.removeAll(al2);
29. System.out.println("After invoking removeAll() method: "+al);
30. //Removing elements on the basis of specified condition
31. al.removeIf(str -> str.contains("Ajay"));   //Here, we are using Lambda expression
32. System.out.println("After invoking removeIf() method: "+al);
33. //Removing all the elements available in the list
34. al.clear();
35. System.out.println("After invoking clear() method: "+al);
36. }
37. }

**Output:**

An initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav]

After invoking remove(index) method: [Ajay, Anuj, Gaurav]

Updated list : [Ajay, Anuj, Gaurav, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav]

After invoking removeIf() method: [Anuj, Gaurav]

After invoking clear() method: []

### **Java ArrayList example of retainAll() method**

1. **import** java.util.\*;
2. **class** ArrayList9{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.retainAll(al2);
12. System.out.println("iterating the elements after retaining the elements of al2");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. }
18. }

**Output:**

iterating the elements after retaining the elements of al2

Ravi

### **Java ArrayList example of isEmpty() method**

1. **import** java.util.\*;
2. **class** ArrayList10{
4. **public** **static** **void** main(String [] args)
5. {
6. ArrayList<String> al=**new** ArrayList<String>();
7. System.out.println("Is ArrayList Empty: "+al.isEmpty());
8. al.add("Ravi");
9. al.add("Vijay");
10. al.add("Ajay");
11. System.out.println("After Insertion");
12. System.out.println("Is ArrayList Empty: "+al.isEmpty());
13. }
14. }

**Output:**

Is ArrayList Empty: true

After Insertion

Is ArrayList Empty: false

### **Java ArrayList Example: Book**

Let's see an ArrayList example where we are adding books to list and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayListExample20 {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample20" \t "_blank)**

**Output:**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# **Java LinkedList class**

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.



### **Java LinkedList Example**

1. **import** java.util.\*;
2. **public** **class** LinkedList1{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> al=**new** LinkedList<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output: Ravi

Vijay

Ravi

Ajay

### **Java LinkedList example to add elements**

Here, we see different ways to add elements.

1. **import** java.util.\*;
2. **public** **class** LinkedList2{
3. **public** **static** **void** main(String args[]){
4. LinkedList<String> ll=**new** LinkedList<String>();
5. System.out.println("Initial list of elements: "+ll);
6. ll.add("Ravi");
7. ll.add("Vijay");
8. ll.add("Ajay");
9. System.out.println("After invoking add(E e) method: "+ll);
10. //Adding an element at the specific position
11. ll.add(1, "Gaurav");
12. System.out.println("After invoking add(int index, E element) method: "+ll);
13. LinkedList<String> ll2=**new** LinkedList<String>();
14. ll2.add("Sonoo");
15. ll2.add("Hanumat");
16. //Adding second list elements to the first list
17. ll.addAll(ll2);
18. System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
19. LinkedList<String> ll3=**new** LinkedList<String>();
20. ll3.add("John");
21. ll3.add("Rahul");
22. //Adding second list elements to the first list at specific position
23. ll.addAll(1, ll3);
24. System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+ll);
25. //Adding an element at the first position
26. ll.addFirst("Lokesh");
27. System.out.println("After invoking addFirst(E e) method: "+ll);
28. //Adding an element at the last position
29. ll.addLast("Harsh");
30. System.out.println("After invoking addLast(E e) method: "+ll);
32. }
33. }

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addFirst(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addLast(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]

### **Java LinkedList example to remove elements**

Here, we see different ways to remove an element.

1. **import** java.util.\*;
2. **public** **class** LinkedList3 {
4. **public** **static** **void** main(String [] args)
5. {
6. LinkedList<String> ll=**new** LinkedList<String>();
7. ll.add("Ravi");
8. ll.add("Vijay");
9. ll.add("Ajay");
10. ll.add("Anuj");
11. ll.add("Gaurav");
12. ll.add("Harsh");
13. ll.add("Virat");
14. ll.add("Gaurav");
15. ll.add("Harsh");
16. ll.add("Amit");
17. System.out.println("Initial list of elements: "+ll);
18. //Removing specific element from arraylist
19. ll.remove("Vijay");
20. System.out.println("After invoking remove(object) method: "+ll);
21. //Removing element on the basis of specific position
22. ll.remove(0);
23. System.out.println("After invoking remove(index) method: "+ll);
24. LinkedList<String> ll2=**new** LinkedList<String>();
25. ll2.add("Ravi");
26. ll2.add("Hanumat");
27. // Adding new elements to arraylist
28. ll.addAll(ll2);
29. System.out.println("Updated list : "+ll);
30. //Removing all the new elements from arraylist
31. ll.removeAll(ll2);
32. System.out.println("After invoking removeAll() method: "+ll);
33. //Removing first element from the list
34. ll.removeFirst();
35. System.out.println("After invoking removeFirst() method: "+ll);
36. //Removing first element from the list
37. ll.removeLast();
38. System.out.println("After invoking removeLast() method: "+ll);
39. //Removing first occurrence of element from the list
40. ll.removeFirstOccurrence("Gaurav");
41. System.out.println("After invoking removeFirstOccurrence() method: "+ll);
42. //Removing last occurrence of element from the list
43. ll.removeLastOccurrence("Harsh");
44. System.out.println("After invoking removeLastOccurrence() method: "+ll);
46. //Removing all the elements available in the list
47. ll.clear();
48. System.out.println("After invoking clear() method: "+ll);
49. }
50. }

Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]

After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]

After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]

After invoking clear() method: []

### **Java LinkedList Example to reverse a list of elements**

1. **import** java.util.\*;
2. **public** **class** LinkedList4{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> ll=**new** LinkedList<String>();
6. ll.add("Ravi");
7. ll.add("Vijay");
8. ll.add("Ajay");
9. //Traversing the list of elements in reverse order
10. Iterator i=ll.descendingIterator();
11. **while**(i.hasNext())
12. {
13. System.out.println(i.next());
14. }
16. }
17. }

Output: Ajay

Vijay

Ravi

### **Java LinkedList Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** LinkedList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a  **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
|  |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

# **Java HashSet**

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.

### **Java HashSet Example**

Let's see a simple example of HashSet. Notice, the elements iterate in an unordered collection.

1. **import** java.util.\*;
2. **class** HashSet1{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet();
6. set.add("One");
7. set.add("Two");
8. set.add("Three");
9. set.add("Four");
10. set.add("Five");
11. Iterator<String> i=set.iterator();
12. **while**(i.hasNext())
13. {
14. System.out.println(i.next());
15. }
16. }
17. }

Five

One

Four

Two

Three

### **Java HashSet example ignoring duplicate elements**

In this example, we see that HashSet doesn't allow duplicate elements.

1. **import** java.util.\*;
2. **class** HashSet2{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Ajay

Vijay

Ravi

### **Java HashSet example to remove elements**

Here, we see different ways to remove an element.

1. **import** java.util.\*;
2. **class** HashSet3{
3. **public** **static** **void** main(String args[]){
4. HashSet<String> set=**new** HashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Arun");
8. set.add("Sumit");
9. System.out.println("An initial list of elements: "+set);
10. //Removing specific element from HashSet
11. set.remove("Ravi");
12. System.out.println("After invoking remove(object) method: "+set);
13. HashSet<String> set1=**new** HashSet<String>();
14. set1.add("Ajay");
15. set1.add("Gaurav");
16. set.addAll(set1);
17. System.out.println("Updated List: "+set);
18. //Removing all the new elements from HashSet
19. set.removeAll(set1);
20. System.out.println("After invoking removeAll() method: "+set);
21. //Removing elements on the basis of specified condition
22. set.removeIf(str->str.contains("Vijay"));
23. System.out.println("After invoking removeIf() method: "+set);
24. //Removing all the elements available in the set
25. set.clear();
26. System.out.println("After invoking clear() method: "+set);
27. }
28. }

An initial list of elements: [Vijay, Ravi, Arun, Sumit]

After invoking remove(object) method: [Vijay, Arun, Sumit]

Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]

After invoking removeAll() method: [Vijay, Arun, Sumit]

After invoking removeIf() method: [Arun, Sumit]

After invoking clear() method: []

### **Java HashSet from another Collection**

1. **import** java.util.\*;
2. **class** HashSet4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();
5. list.add("Ravi");
6. list.add("Vijay");
7. list.add("Ajay");
9. HashSet<String> set=**new** HashSet(list);
10. set.add("Gaurav");
11. Iterator<String> i=set.iterator();
12. **while**(i.hasNext())
13. {
14. System.out.println(i.next());
15. }
16. }
17. }

Vijay

Ravi

Gaurav

Ajay

### **Java HashSet Example: Book**

Let's see a HashSet example where we are adding books to set and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** HashSetExample {
15. **public** **static** **void** main(String[] args) {
16. HashSet<Book> set=**new** HashSet<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to HashSet
22. set.add(b1);
23. set.add(b2);
24. set.add(b3);
25. //Traversing HashSet
26. **for**(Book b:set){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# **Java LinkedHashSet class**

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operation and permits null elements.
* Java LinkedHashSet class maintains insertion order.

### **Java LinkedHashSet Example**

Let's see a simple example of Java LinkedHashSet class. Here you can notice that the elements iterate in insertion order.

1. **import** java.util.\*;
2. **class** LinkedHashSet1{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. LinkedHashSet<String> set=**new** LinkedHashSet();
6. set.add("One");
7. set.add("Two");
8. set.add("Three");
9. set.add("Four");
10. set.add("Five");
11. Iterator<String> i=set.iterator();
12. **while**(i.hasNext())
13. {
14. System.out.println(i.next());
15. }
16. }
17. }

One

Two

Three

Four

Five

### **Java LinkedHashSet example ignoring duplicate Elements**

1. **import** java.util.\*;
2. **class** LinkedHashSet2{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> al=**new** LinkedHashSet<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Ravi

Vijay

Ajay

### **Java LinkedHashSet Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedHashSetExample {
15. **public** **static** **void** main(String[] args) {
16. LinkedHashSet<Book> hs=**new** LinkedHashSet<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to hash table
22. hs.add(b1);
23. hs.add(b2);
24. hs.add(b3);
25. //Traversing hash table
26. **for**(Book b:hs){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# **Java TreeSet class**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.

### **Java TreeSet Examples**

### **Java TreeSet Example 1:**

Let's see a simple example of Java TreeSet.

1. **import** java.util.\*;
2. **class** TreeSet1{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> al=**new** TreeSet<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection11" \t "_blank)**

Output:

Ajay

Ravi

Vijay

### **Java TreeSet Example 2:**

Let's see an example of traversing elements in descending order.

1. **import** java.util.\*;
2. **class** TreeSet2{
3. **public** **static** **void** main(String args[]){
4. TreeSet<String> set=**new** TreeSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ajay");
8. System.out.println("Traversing element through Iterator in descending order");
9. Iterator i=set.descendingIterator();
10. **while**(i.hasNext())
11. {
12. System.out.println(i.next());
13. }
15. }
16. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection11" \t "_blank)**

Output:

Traversing element through Iterator in descending order

Vijay

Ravi

Ajay

Traversing element through NavigableSet in descending order

Vijay

Ravi

Ajay

### **Java TreeSet Example 3:**

Let's see an example to retrieve and remove the highest and lowest Value.

1. **import** java.util.\*;
2. **class** TreeSet3{
3. **public** **static** **void** main(String args[]){
4. TreeSet<Integer> set=**new** TreeSet<Integer>();
5. set.add(24);
6. set.add(66);
7. set.add(12);
8. set.add(15);
9. System.out.println("Highest Value: "+set.pollFirst());
10. System.out.println("Lowest Value: "+set.pollLast());
11. }
12. }

Output:

Highest Value: 12

Lowest Value: 66

### **Java TreeSet Example 4:**

In this example, we perform various NavigableSet operations.

1. **import** java.util.\*;
2. **class** TreeSet4{
3. **public** **static** **void** main(String args[]){
4. TreeSet<String> set=**new** TreeSet<String>();
5. set.add("A");
6. set.add("B");
7. set.add("C");
8. set.add("D");
9. set.add("E");
10. System.out.println("Initial Set: "+set);
12. System.out.println("Reverse Set: "+set.descendingSet());
14. System.out.println("Head Set: "+set.headSet("C", **true**));
16. System.out.println("SubSet: "+set.subSet("A", **false**, "E", **true**));
18. System.out.println("TailSet: "+set.tailSet("C", **false**));
19. }
20. }

Output:

Initial Set: [A, B, C, D, E]

Reverse Set: [E, D, C, B, A]

Head Set: [A, B, C]

SubSet: [B, C, D, E]

TailSet: [D, E]

### **Java TreeSet Example 4:**

In this example, we perform various SortedSetSet operations.

1. **import** java.util.\*;
2. **class** TreeSet4{
3. **public** **static** **void** main(String args[]){
4. TreeSet<String> set=**new** TreeSet<String>();
5. set.add("A");
6. set.add("B");
7. set.add("C");
8. set.add("D");
9. set.add("E");
11. System.out.println("Intial Set: "+set);
13. System.out.println("Head Set: "+set.headSet("C"));
15. System.out.println("SubSet: "+set.subSet("A", "E"));
17. System.out.println("TailSet: "+set.tailSet("C"));
18. }
19. }

Output:

Intial Set: [A, B, C, D, E]

Head Set: [A, B]

SubSet: [A, B, C, D]

TailSet: [C, D, E]

### **Java TreeSet Example: Book**

Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of a Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement the Comparable interface.

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. **public** **int** compareTo(Book b) {
14. **if**(id>b.id){
15. **return** 1;
16. }**else** **if**(id<b.id){
17. **return** -1;
18. }**else**{
19. **return** 0;
20. }
21. }
22. }
23. **public** **class** TreeSetExample {
24. **public** **static** **void** main(String[] args) {
25. Set<Book> set=**new** TreeSet<Book>();
26. //Creating Books
27. Book b1=**new** Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=**new** Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=**new** Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to TreeSet
31. set.add(b1);
32. set.add(b2);
33. set.add(b3);
34. //Traversing TreeSet
35. **for**(Book b:set){
36. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
37. }
38. }
39. }

Output:

101 Data Communications & Networking Forouzan Mc Graw Hill 4

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6