

Java Tutorial



Our core Java programming tutorial is designed for students and working professionals. Java is an **object-oriented**, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

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

Java Example

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

Simple.java

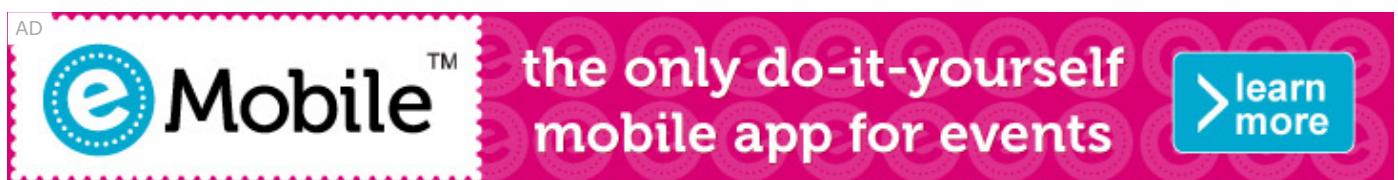
```
class Simple{
    public static void main(String args[]){
        System.out.println("Hello Java");
    }
}
```

[Test it Now](#)

Application

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

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



Types of Java Applications

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

1) Standalone Application

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

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, **Servlet**, **JSP**, **Struts**, **Spring**, **Hibernate**, **JSF**, 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** 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**, 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**, 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.

AD

Prerequisite

To learn Java, you must have the basic knowledge of C/C++ programming language.

Audience

Our Java programming tutorial is designed to help beginners and professionals.

Problem

We assure that you will not find any problem in this Java tutorial. However, if there is any mistake, please post the problem in the contact form.

Do You Know?

- What is the difference between JRE and JVM?
- What is the purpose of JIT compiler?
- Can we save the java source file without any name?
- Why java uses the concept of Unicode system?

What will we learn in Basics of Java?

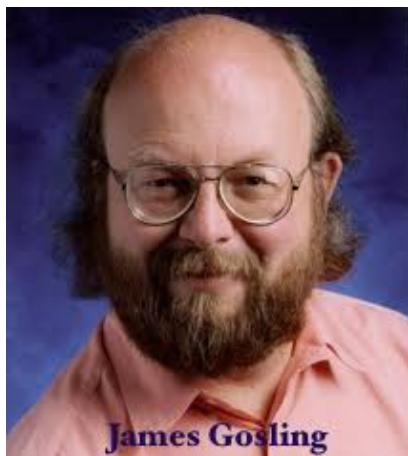
- History of Java
- Features of Java
- Hello Java Program
- Program Internal
- How to set path?
- Difference between JDK,JRE and JVM
- Internal Details of JVM
- Variable and Data Type
- Unicode System
- Operators

Next →

History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

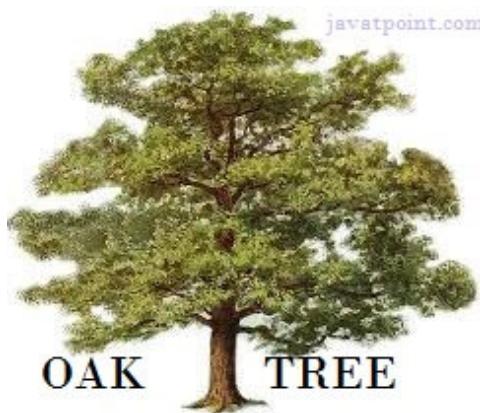
The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". **Java** was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.



Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

- 1) **James Gosling, Mike Sheridan, and Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- 2) Initially it was designed for small, **embedded systems** in electronic appliances like set-top boxes.
- 3) Firstly, it was called "**Greentalk**" by James Gosling, and the file extension was .gt.
- 4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java was named as "Oak"?



5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as "**Java**" because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

7) Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at **Sun Microsystems** (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

History of Java



Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

AD

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)

Since Java SE 8 release, the Oracle corporation follows a pattern in which every even version is released in March month and an odd version released in September month.

More Details on Java Versions.

← Prev

Next →

AD



For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

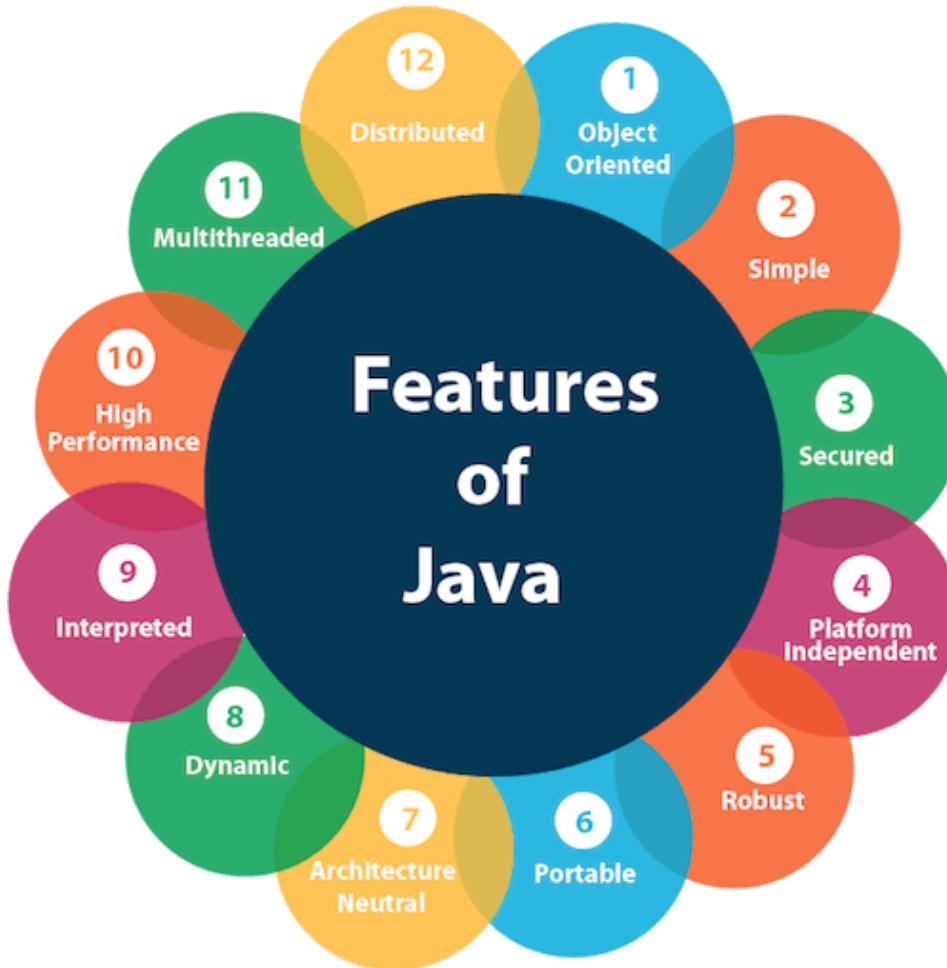
Help Others, Please Share



Features of Java

The primary objective of **Java programming** 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
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted

9. High Performance

10. Multithreaded

11. Distributed

12. Dynamic

Features of Java - Javatpoint



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** 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.

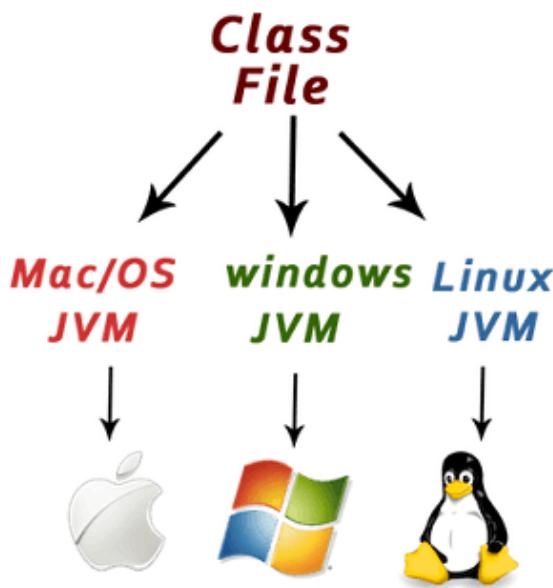
Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object

2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent



Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

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

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

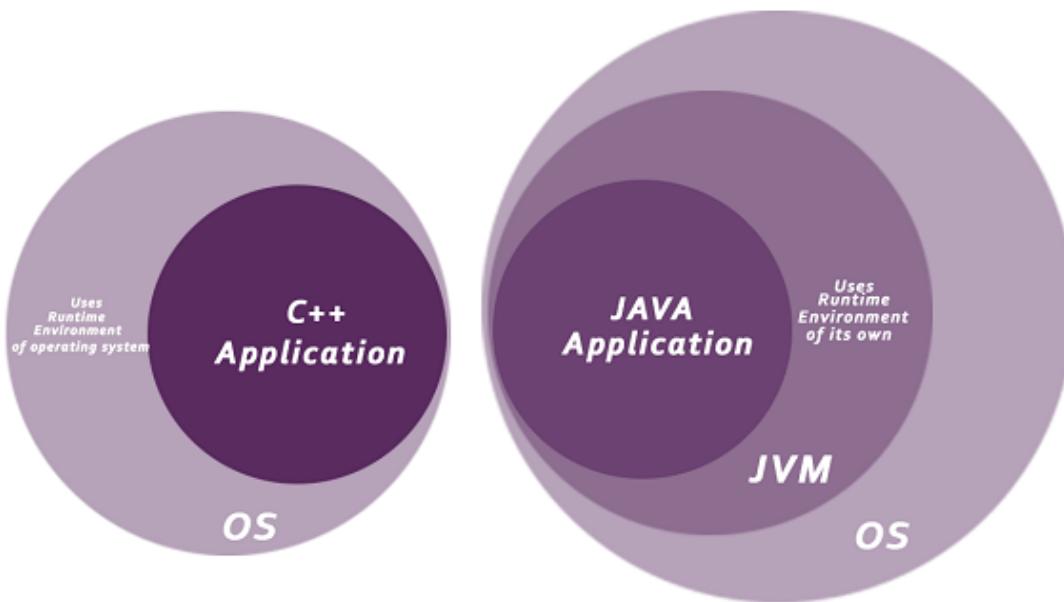
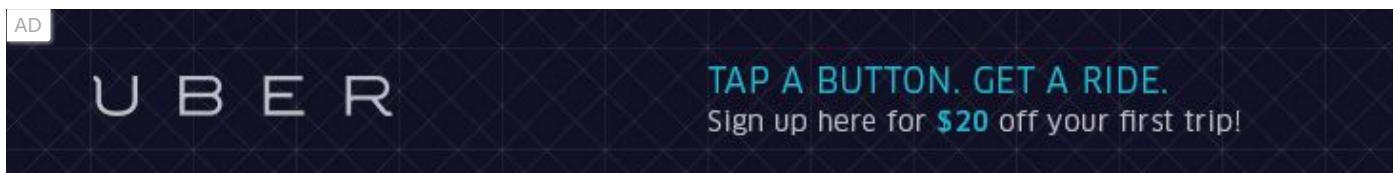
1. Runtime Environment
2. API(Application Programming Interface)

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

Secured

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

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



- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access rights to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

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

Robust

The English meaning 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.

AD

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++.

Java supports dynamic compilation and automatic memory management (garbage collection).

← Prev

Next →

AD



For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



C++ vs Java

There are many differences and similarities between the **C++ programming** language and **Java**. A list of top differences between C++ and Java are given below:

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java .
Operator Overloading	C++ supports operator overloading .	Java doesn't support operator overloading.
Pointers	C++ supports pointers . You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.

Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (<code>/** ... */</code>) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support <code>>>></code> operator.	Java supports unsigned right shift <code>>>></code> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like <code>>></code> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.

Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.
------------------------	--	---

Note

- Java doesn't support default arguments like C++.
- Java does not support header files like C++. Java uses the import keyword to include different classes and methods.

C++ Program Example

File: main.cpp

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello C++ Programming";
    return 0;
}
```

Output:

```
Hello C++ Programming
```

Java Program Example

File: Simple.java

```
class Simple{
    public static void main(String args[]){
        System.out.println("Hello Java");
    }
}
```

Output:

```
Hello Java
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

First Java Program | Hello World Example

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.javatpoint.com/download-jdk) and install it.
- Set path of the jdk/bin directory. <http://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:

```
class Simple{  
    public static void main(String args[]){  
        System.out.println("Hello Java");  
    }  
}
```

Test it Now

Save the above file as Simple.java.

To compile:

javac Simple.java

To execute:

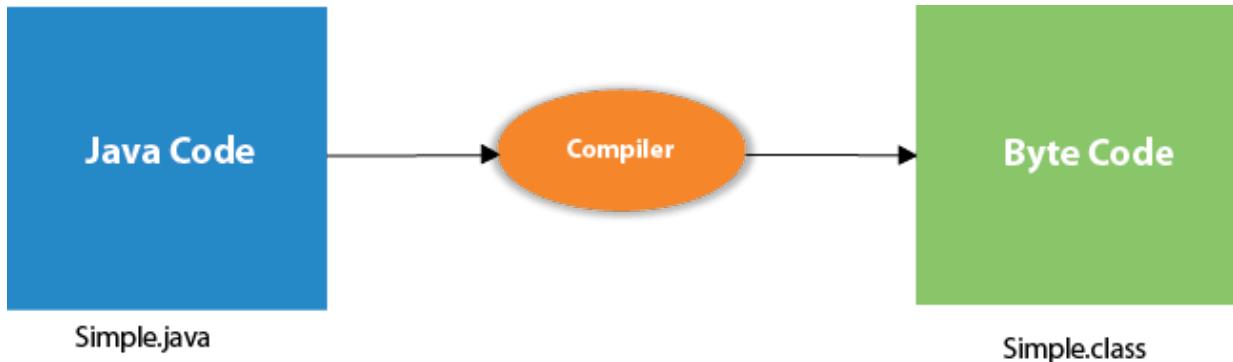
java Simple

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().

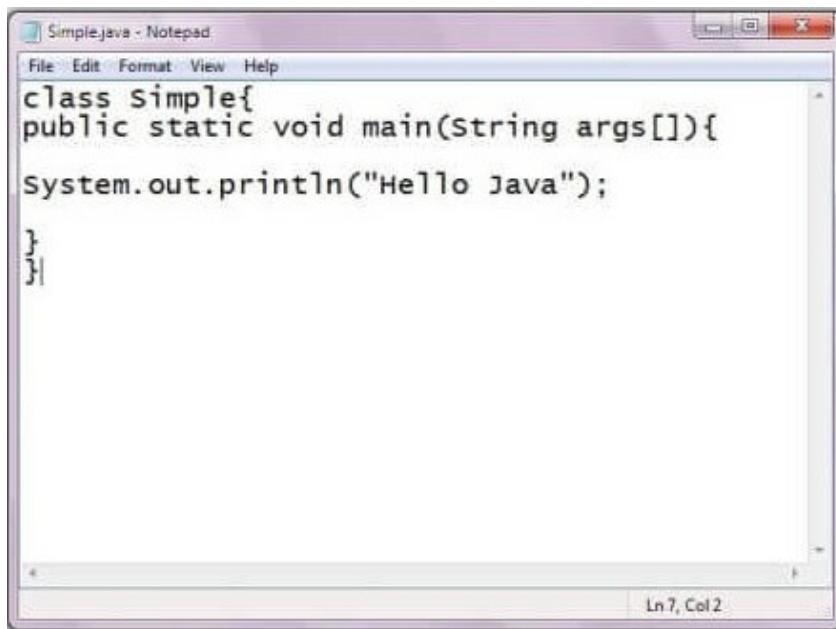
- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that 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**. 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()** statement in the coming section.



Hello Java Program for Beginners



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 shown below:

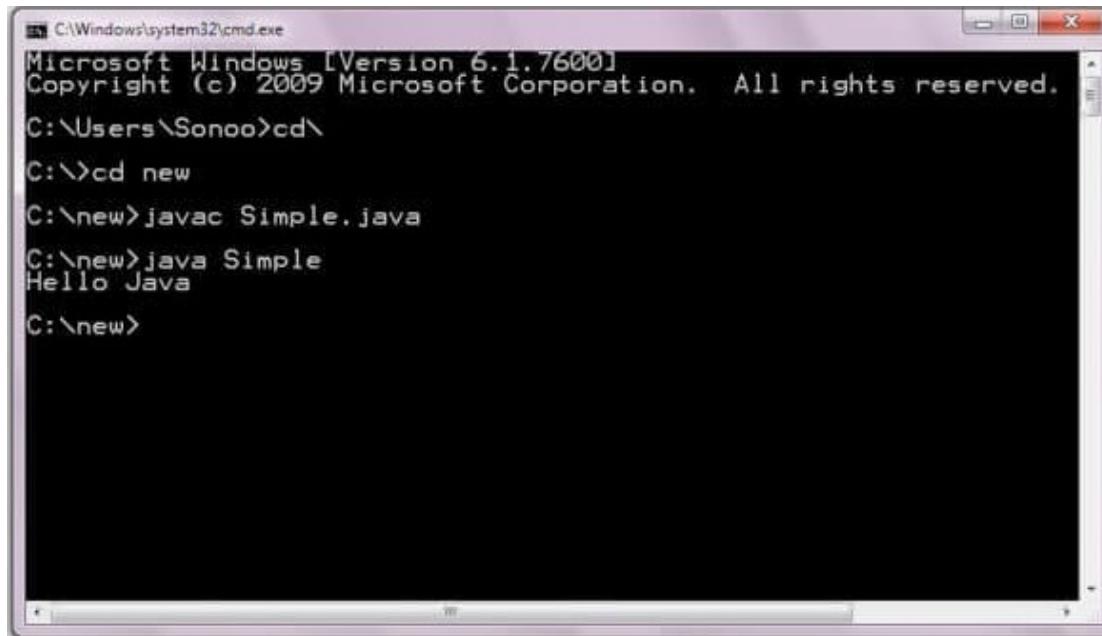


A screenshot of a Windows Notepad window titled "Simple.java - Notepad". The window contains the following Java code:

```
class Simple{
public static void main(String args[]){
System.out.println("Hello Java");
}
}
```

The status bar at the bottom right shows "Ln 7, Col 2".

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:



A screenshot of a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window shows the following text:
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
C:\new>java Simple
Hello Java
C:\new>

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.

```
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.

AD

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

```
public static void main(String[] args)  
public static void main(String []args)  
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.

```
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.



```
class A{  
    static public void main(String... args){  
        System.out.println("hello java4");  
    }  
};
```

Valid Java main() method signature

```
public static void main(String[] args)  
public static void main(String []args)  
public static void main(String args[])  
public static void main(String... args)  
static public void main(String[] args)  
public static final void main(String[] args)  
final public static void main(String[] args)  
final strictfp public static void main(String[] args)
```

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Invalid Java main() method signature

```
public void main(String[] args)
static void main(String[] args)
public void static main(String[] args)
abstract 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](#).

A screenshot of a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window shows the following text:

```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\new>
```

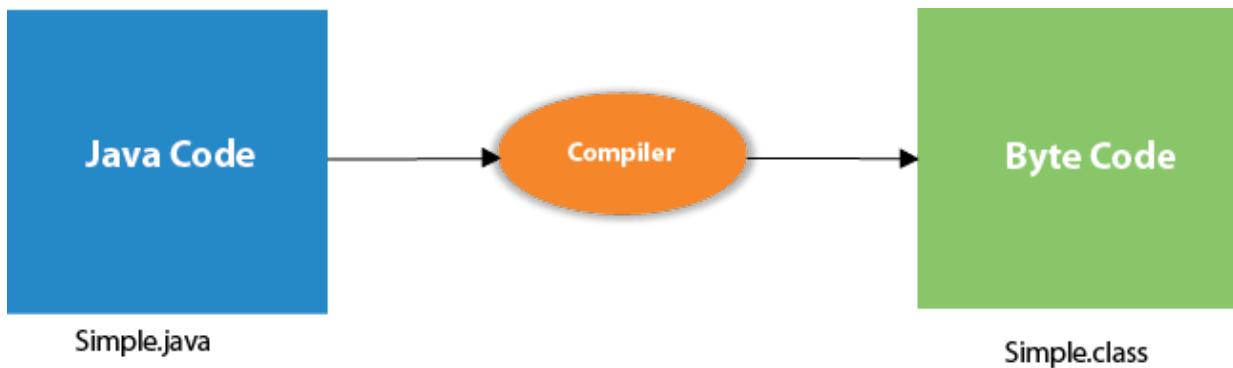
The window has a standard Windows title bar and a scroll bar on the right side.[← Prev](#)[Next →](#)

Internal Details of Hello Java Program

In the previous section, we have created Java Hello World program and learn how to compile and run a Java program. In this section, we are going to learn, what happens while we compile and run the Java program. Moreover, we will see some questions based on the first program.

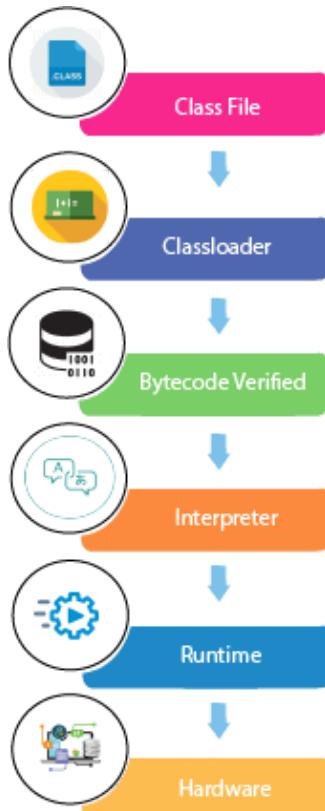
What happens at compile time?

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:



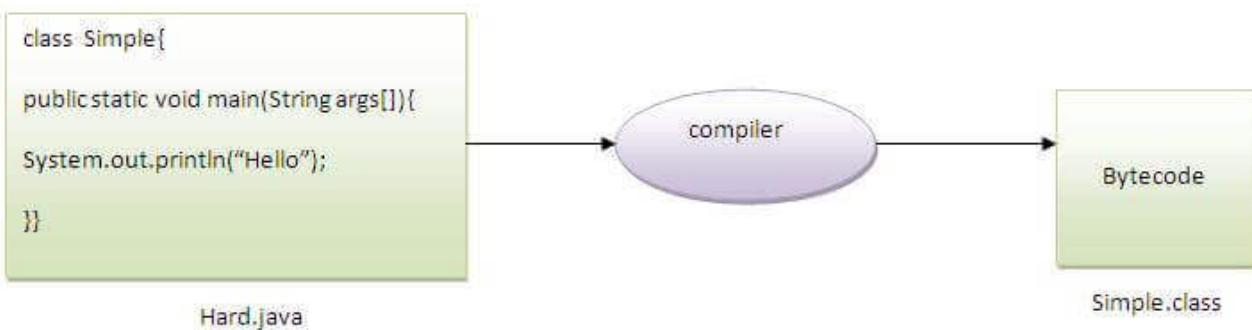
Classloader: It is the subsystem of JVM that is used to load class files.

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

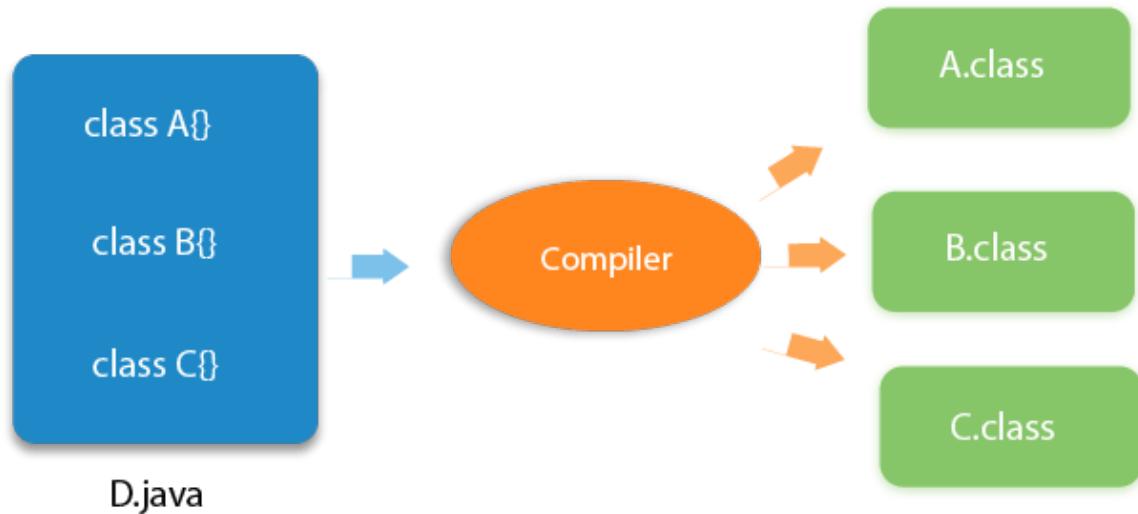
To execute:

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:



← Prev

Next →

AD

[Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



How to set path in Java

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:

1. Temporary
2. Permanent

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

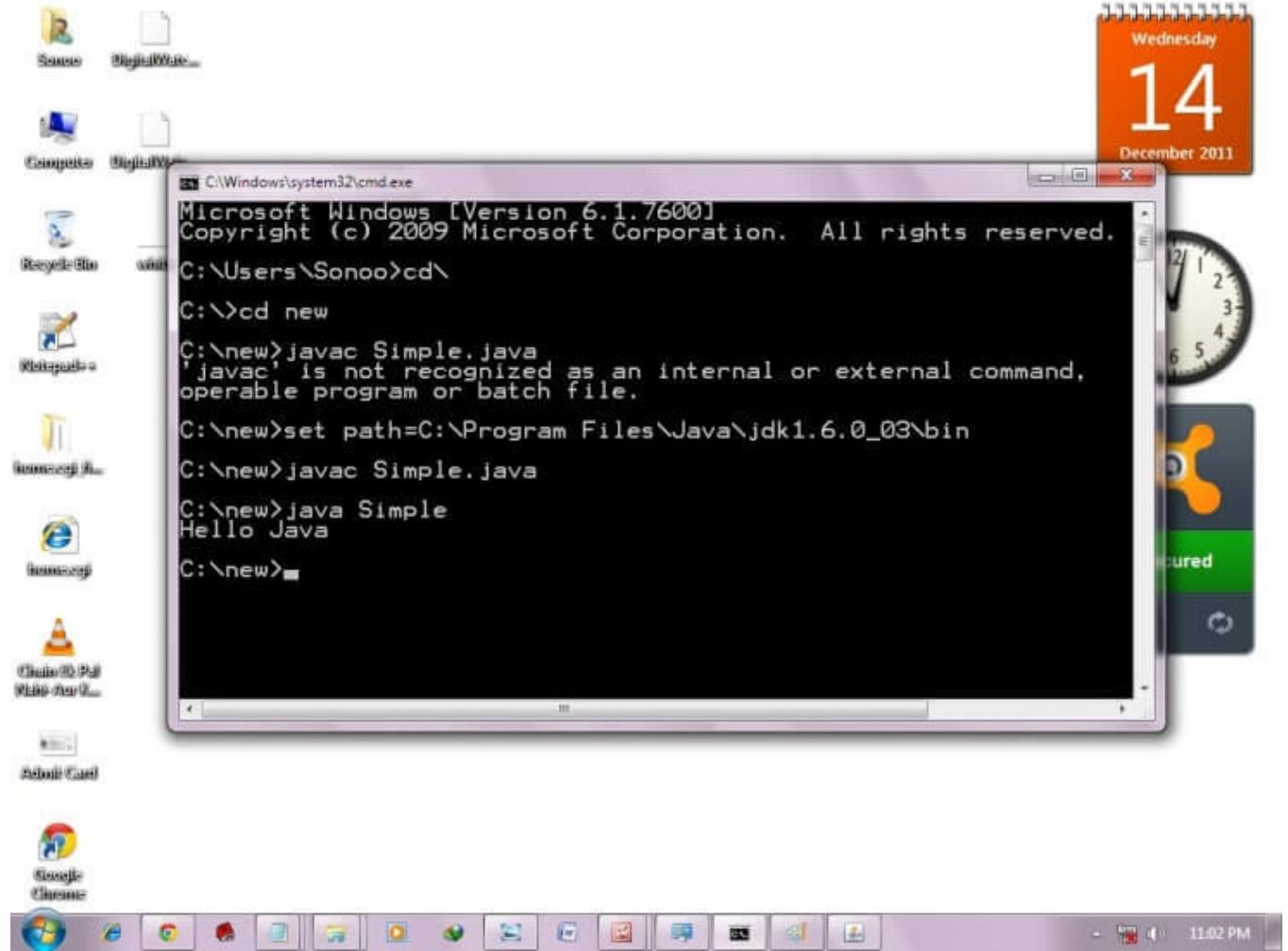
To set the temporary path of JDK, you need to follow the following steps:

- o Open the command prompt
- o Copy the path of the JDK/bin directory
- o Write in command prompt: set path=copied_path

For Example:

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

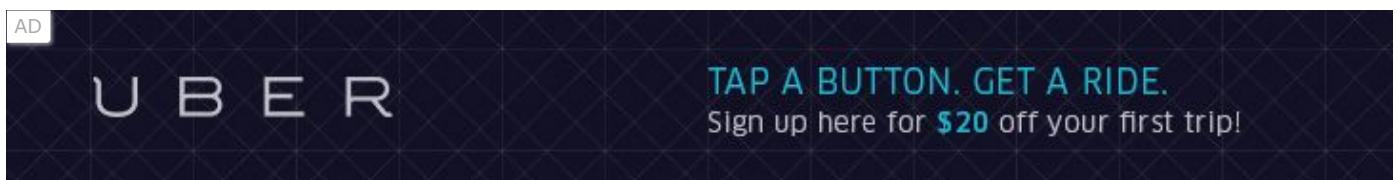
Let's see it in the figure given below:



2) How to set Permanent Path of JDK in Windows

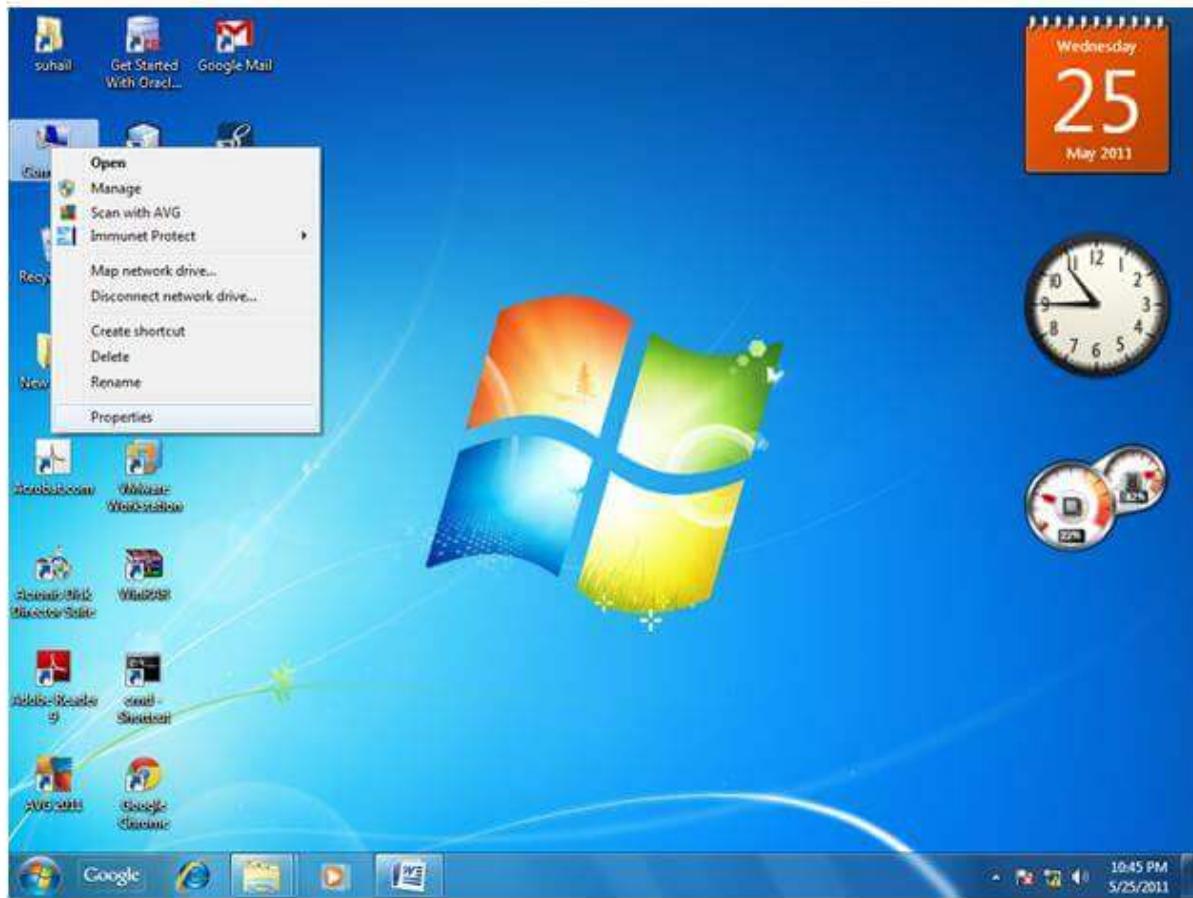
For setting the permanent path of JDK, you need to follow these steps:

- 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:

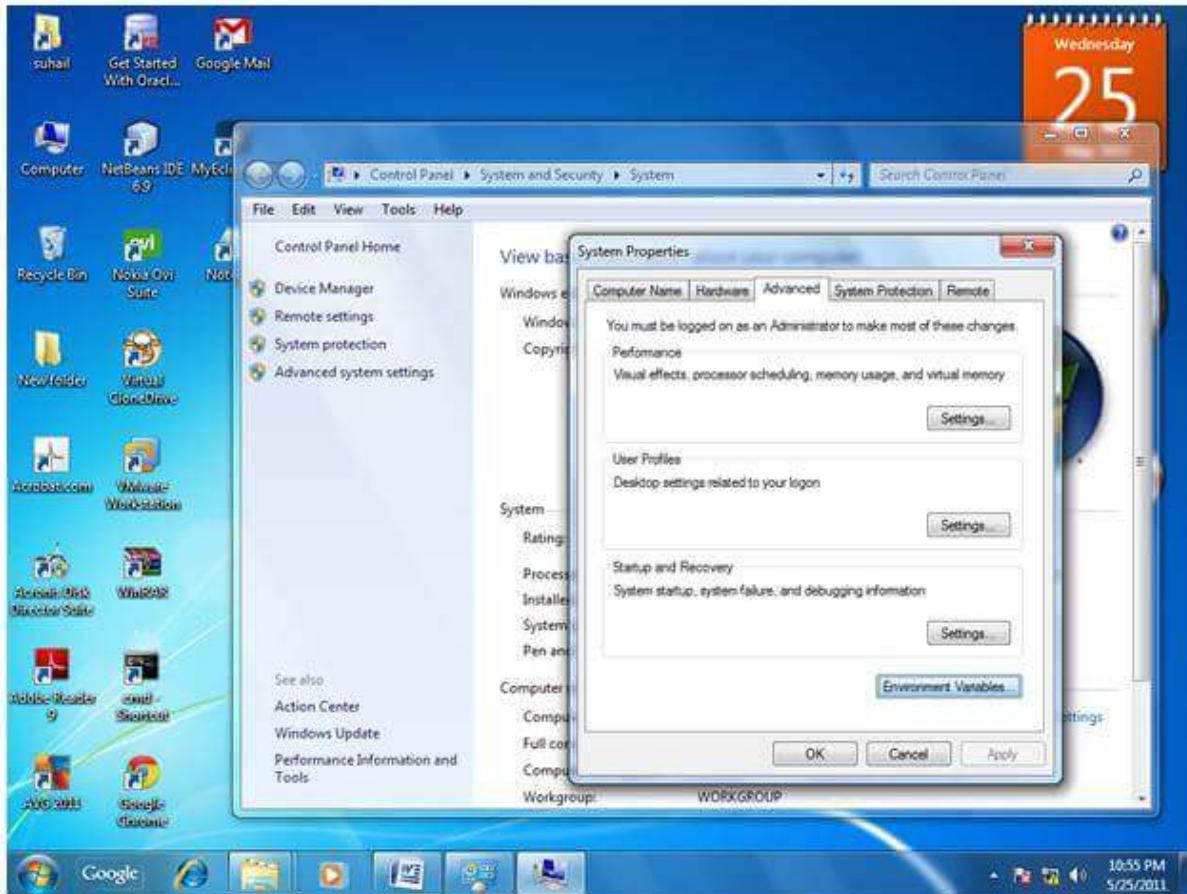
1) Go to MyComputer properties



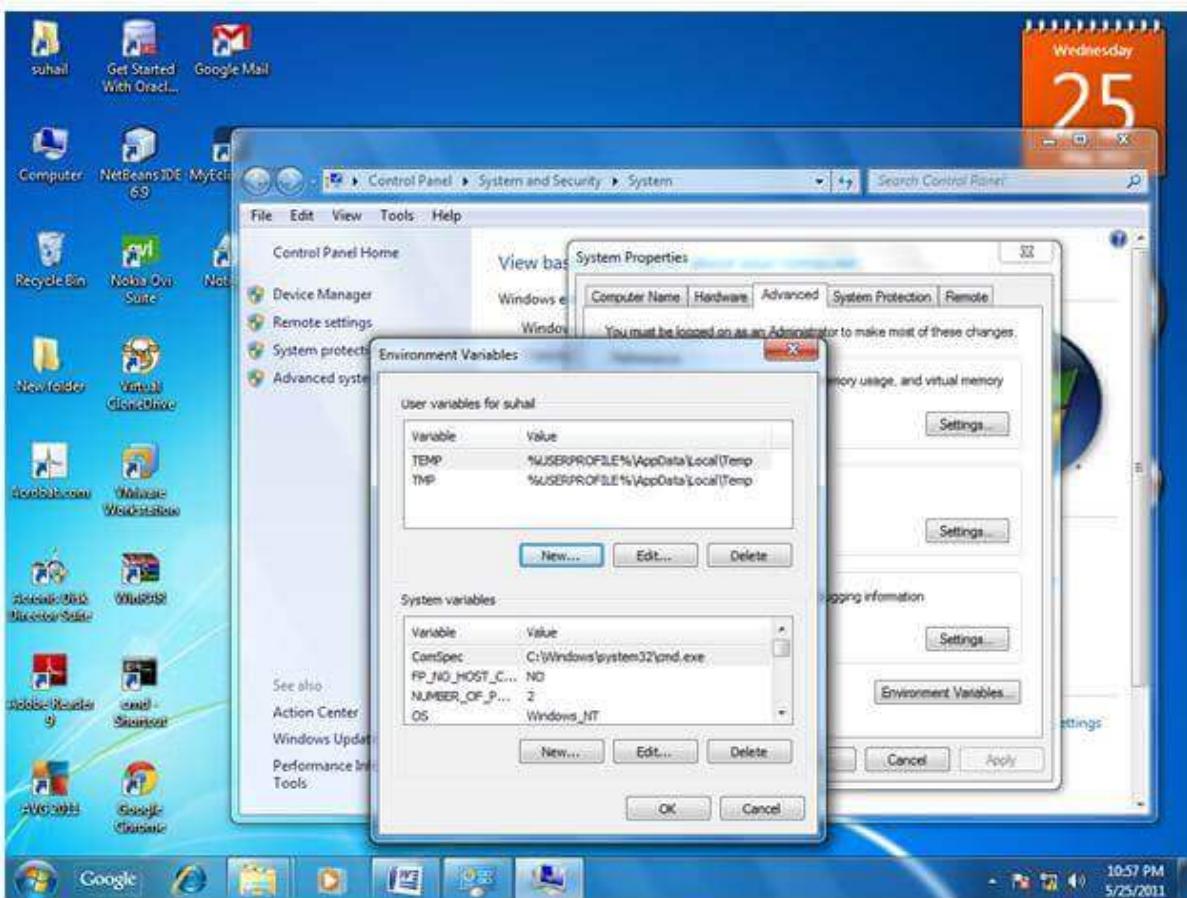
2) Click on the advanced tab



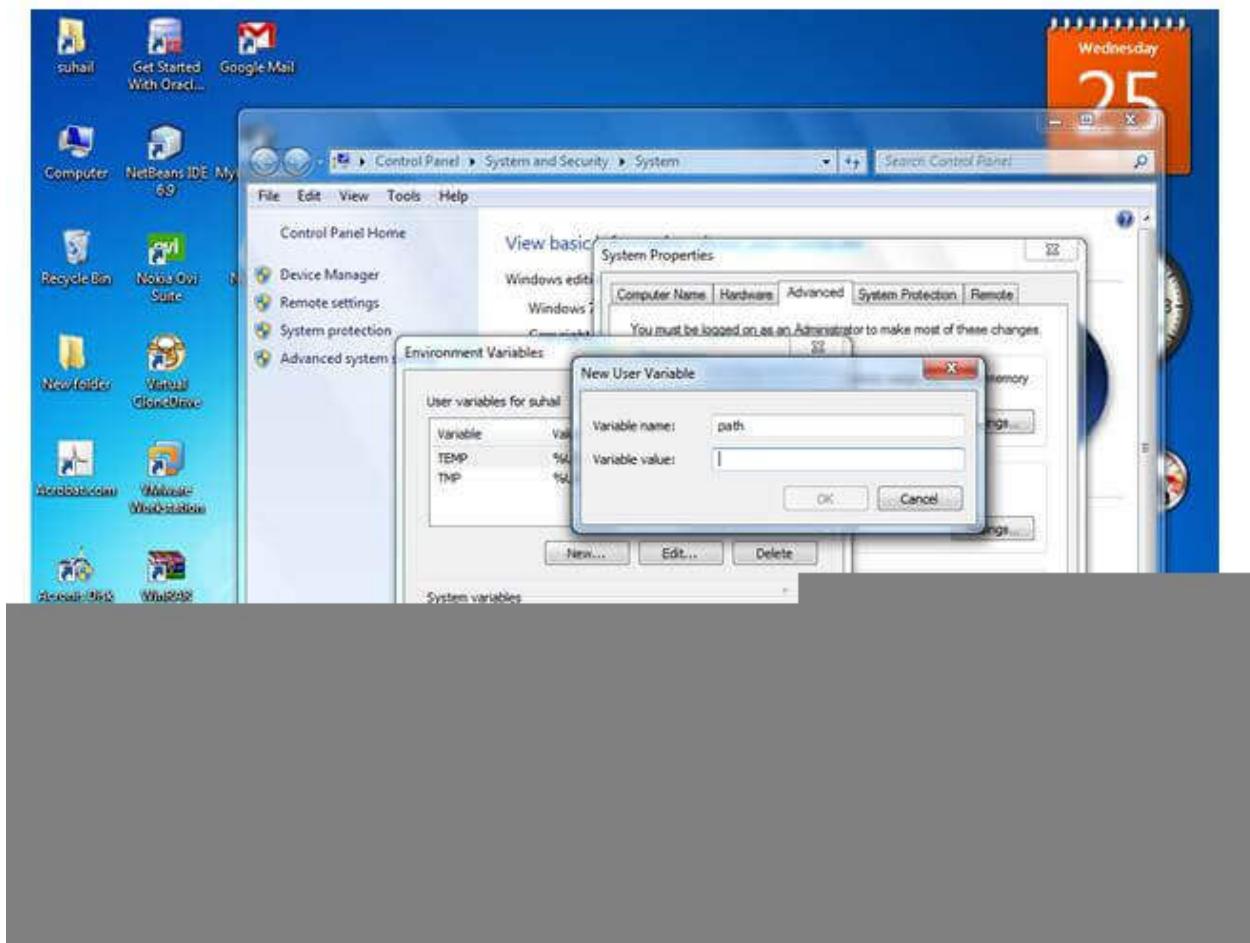
3) Click on environment variables



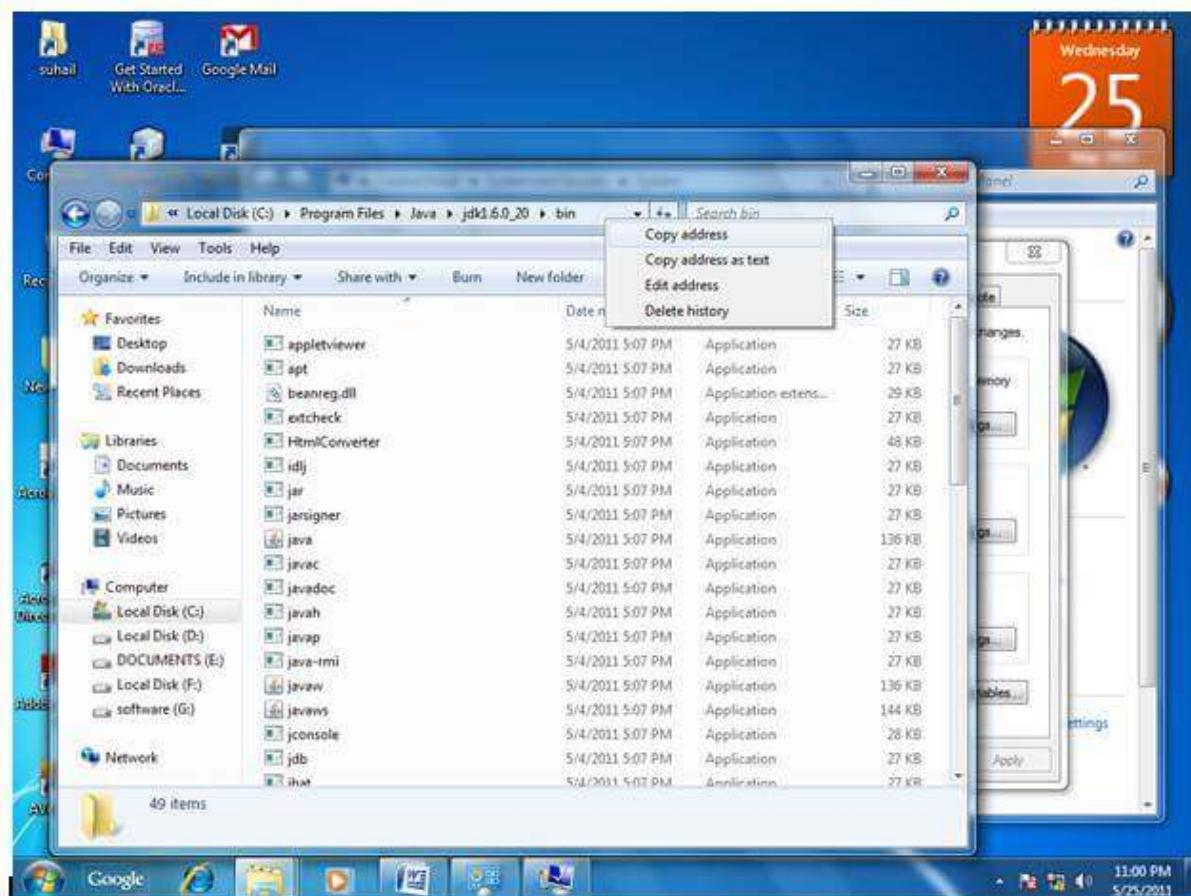
4) Click on the new tab of user variables



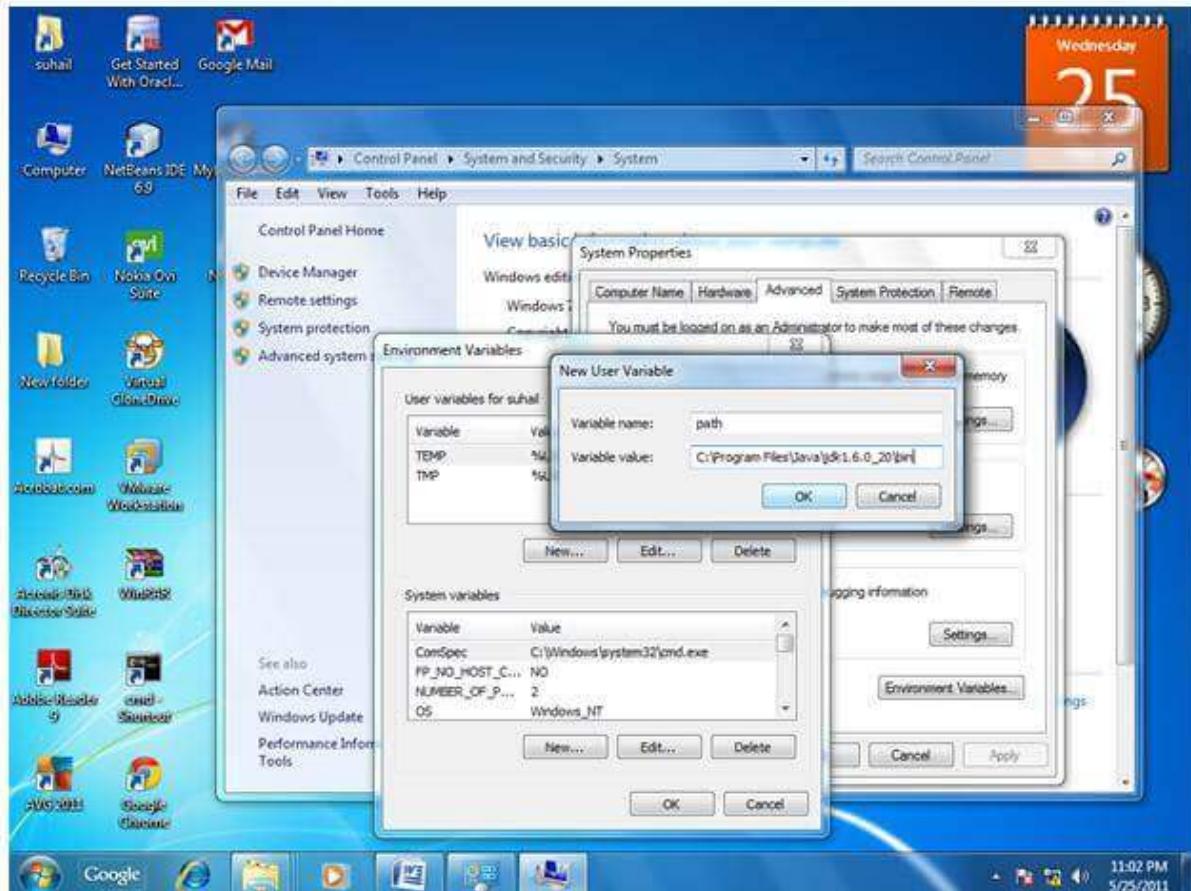
5) Write the path in the variable name



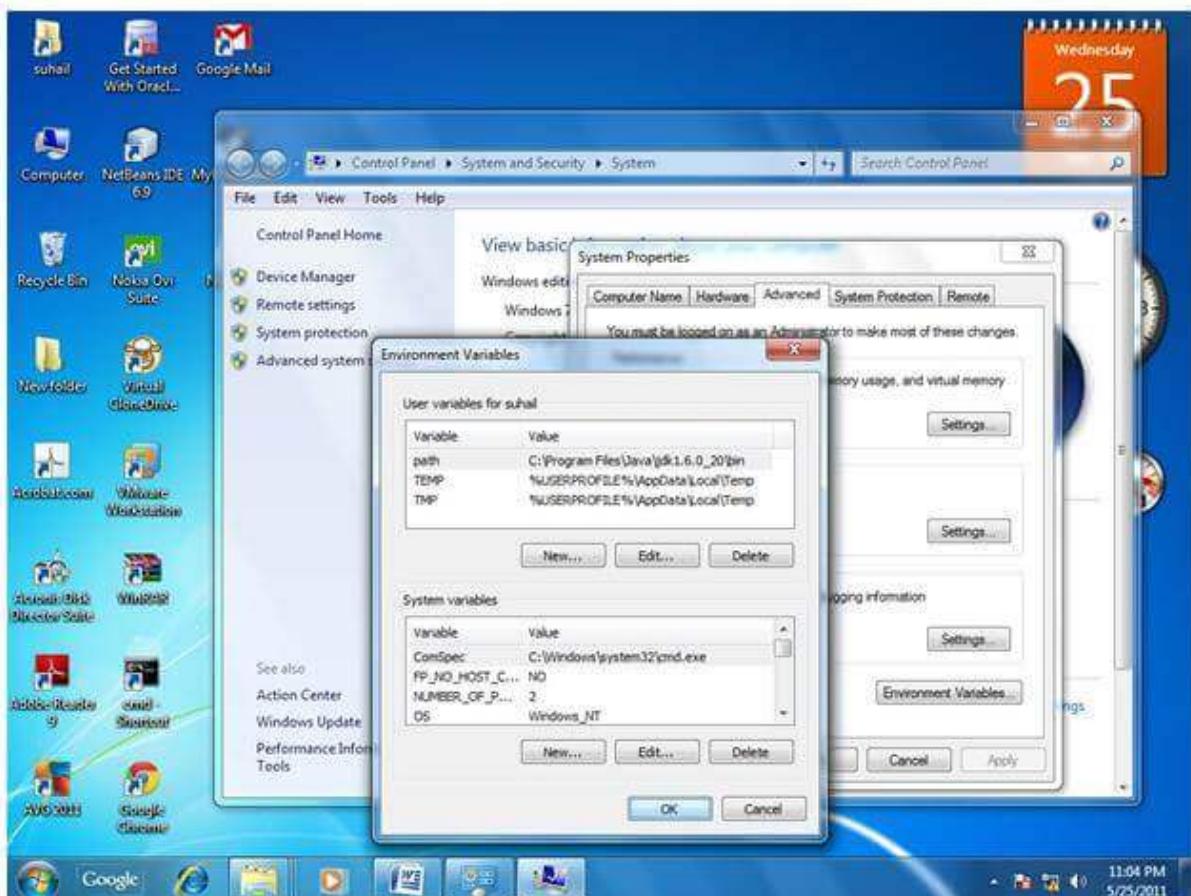
6) Copy the path of bin folder



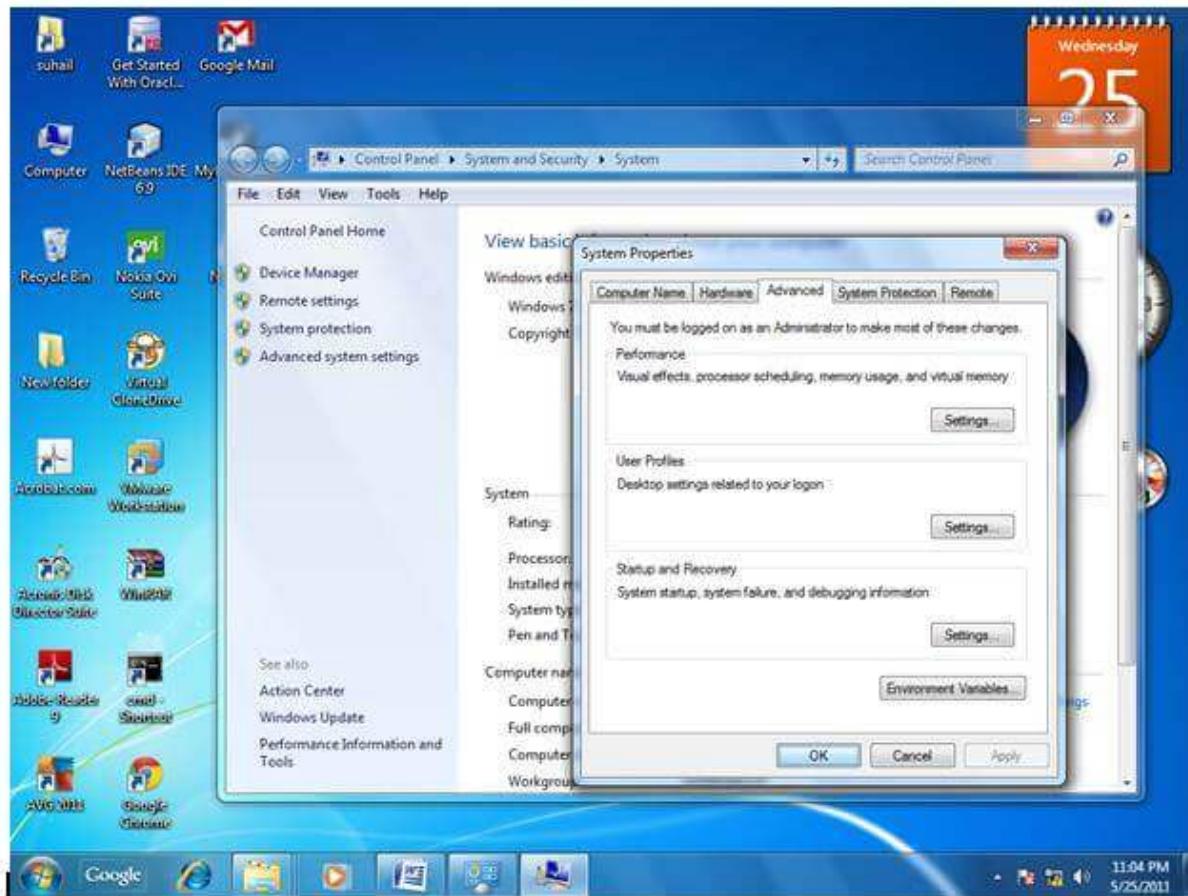
7) Paste path of bin folder in the variable value



8) Click on ok button



9) Click on ok button



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).

How to set path of Java on Windows



You may also like:

[How to set classpath in Java](#)

← Prev

Next →

AD

[For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Difference between JDK, JRE, and JVM

We must understand the differences between JDK, JRE, and JVM before proceeding further to [Java](#). See the brief overview of JVM here.

If you want to get the detailed knowledge of Java Virtual Machine, move to the next page. Firstly, let's see the differences between the JDK, JRE, and JVM.

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each [OS](#) is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

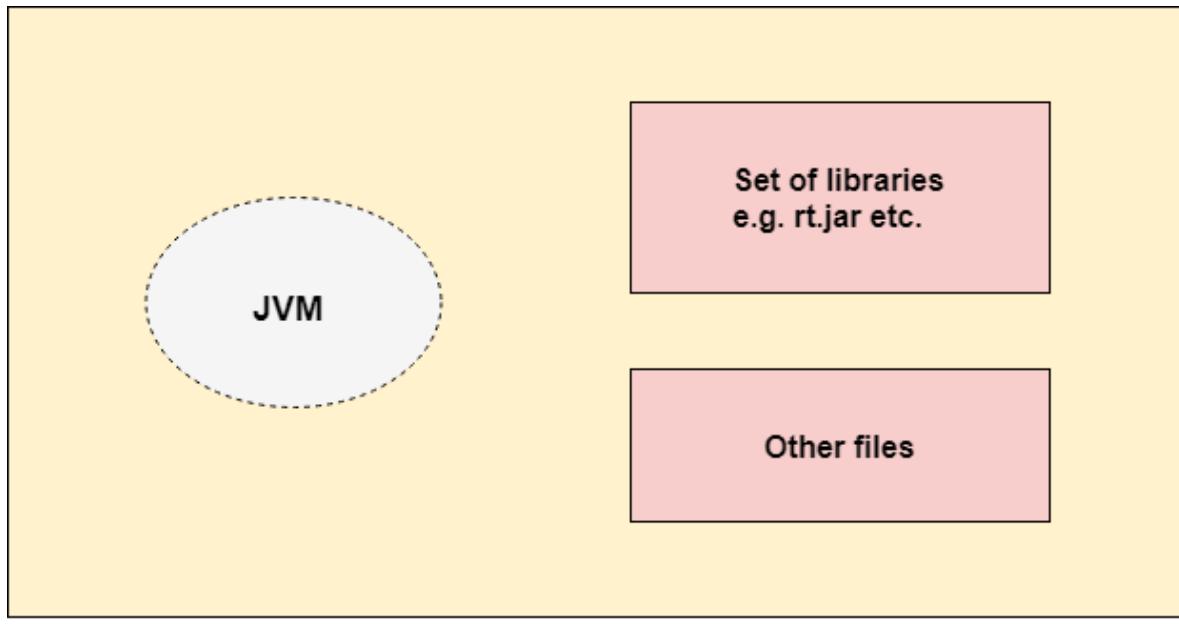
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

[More Details.](#)

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JRE

JDK

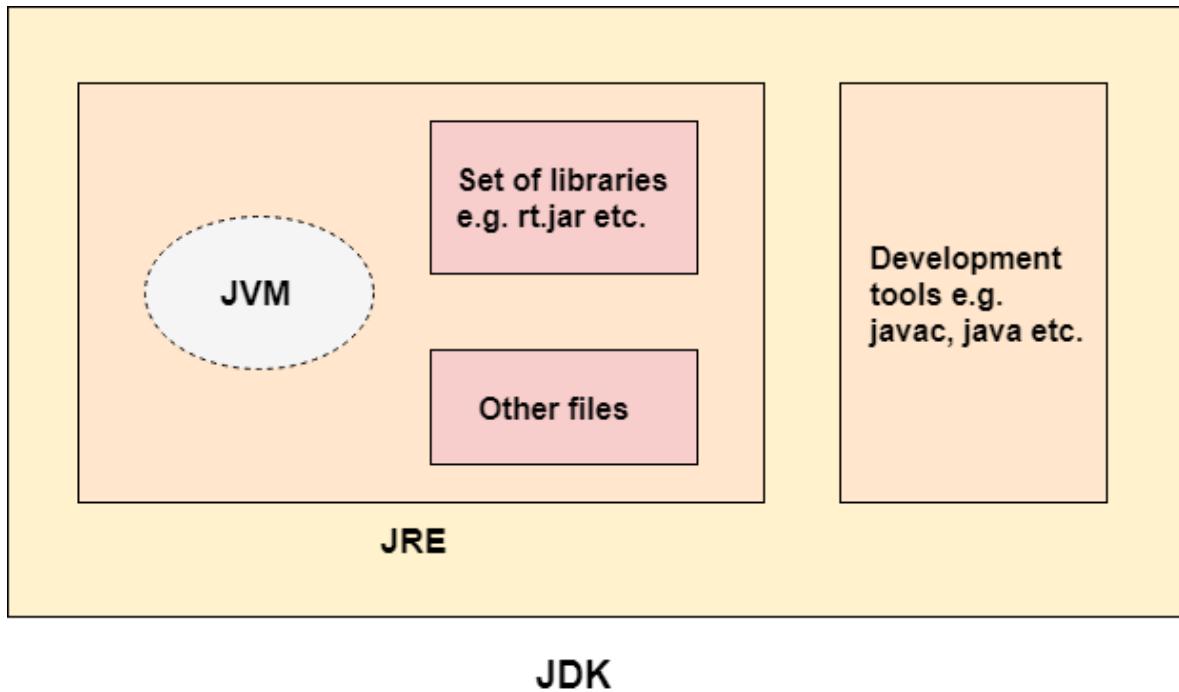
JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and **applets**. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform



The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



[More Details.](#)

Reference Video

0:00 / 5:29

← Prev

Next →

AD

JVM (Java Virtual Machine) Architecture

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

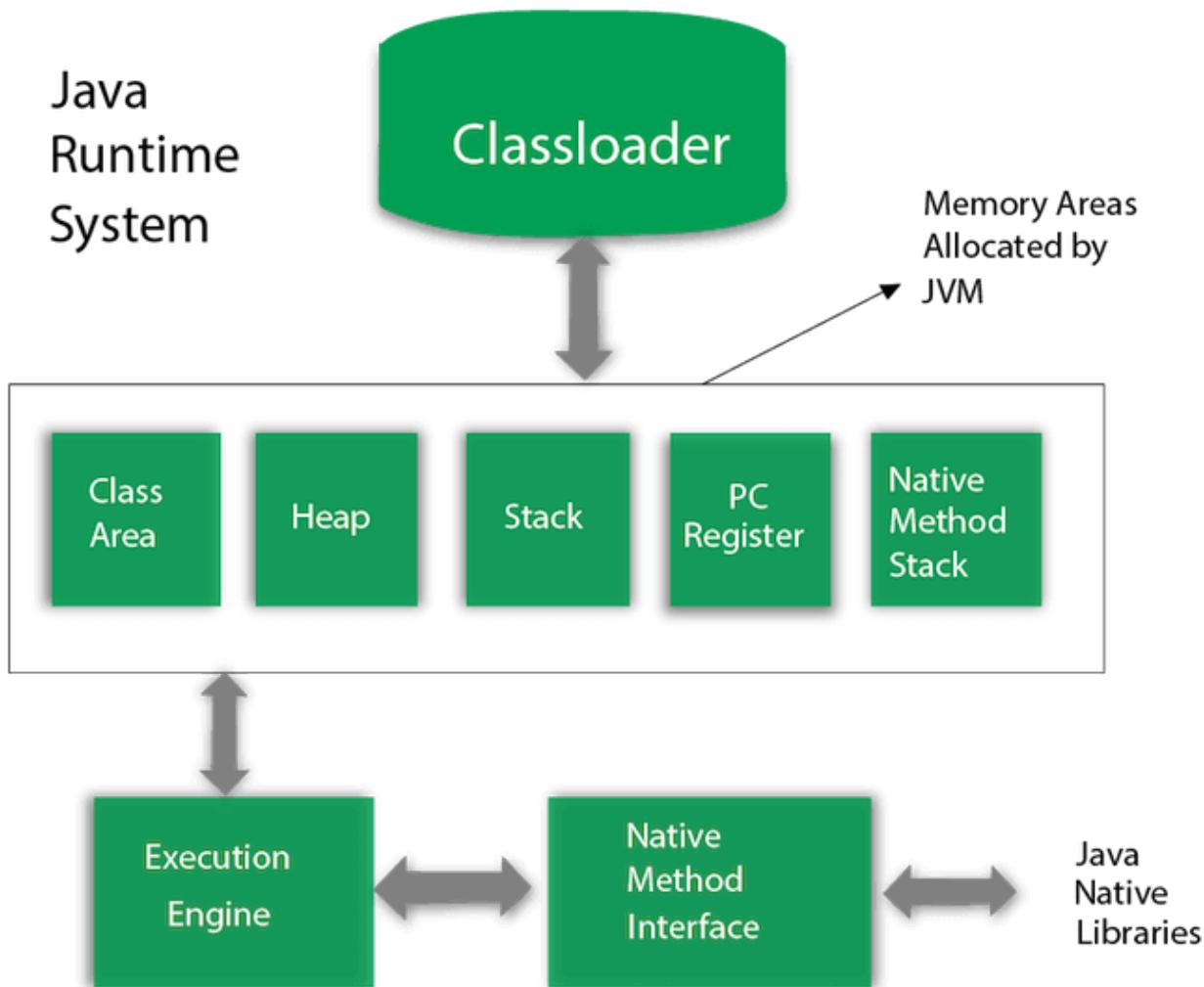
JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

AD

JVM Architecture

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1) Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

- Bootstrap ClassLoader:** This is the first classloader which is the super class of Extension classloader. It loads the `rt.jar` file which contains all class files of Java Standard Edition like `java.lang` package classes, `java.net` package classes, `java.util` package classes, `java.io` package classes, `java.sql` package classes etc.
- Extension ClassLoader:** This is the child classloader of Bootstrap and parent classloader of System classloader. It loads the jar files located inside `$JAVA_HOME/jre/lib/ext` directory.
- System/Application ClassLoader:** This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can

change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

```
//Let's see an example to print the classloader name  
public class ClassLoaderExample  
{  
    public static void main(String[] args)  
    {  
        // Let's print the classloader name of current class.  
        //Application/System classloader will load this class  
        Class c=ClassLoaderExample.class;  
        System.out.println(c.getClassLoader());  
        //If we print the classloader name of String, it will print null because it is an  
        //in-built class which is found in rt.jar, so it is loaded by Bootstrap classloader  
        System.out.println(String.class.getClassLoader());  
    }  
}
```

Test it Now

Output:

```
sun.misc.Launcher$AppClassLoader@4e0e2f2a  
null
```

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

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.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

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:

AD

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.

8) Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

Java Variables

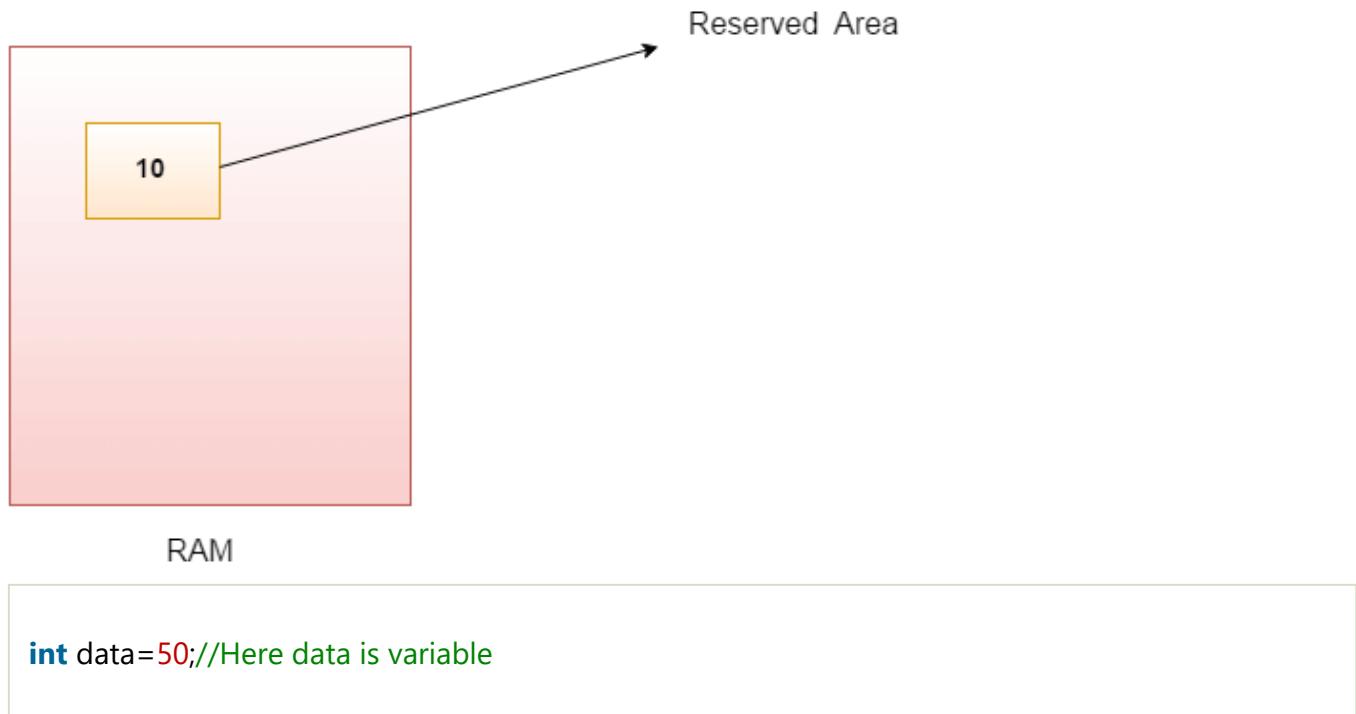
A variable is a container which holds the value while the **Java program** is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of **data types in Java**: primitive and non-primitive.

Variable

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

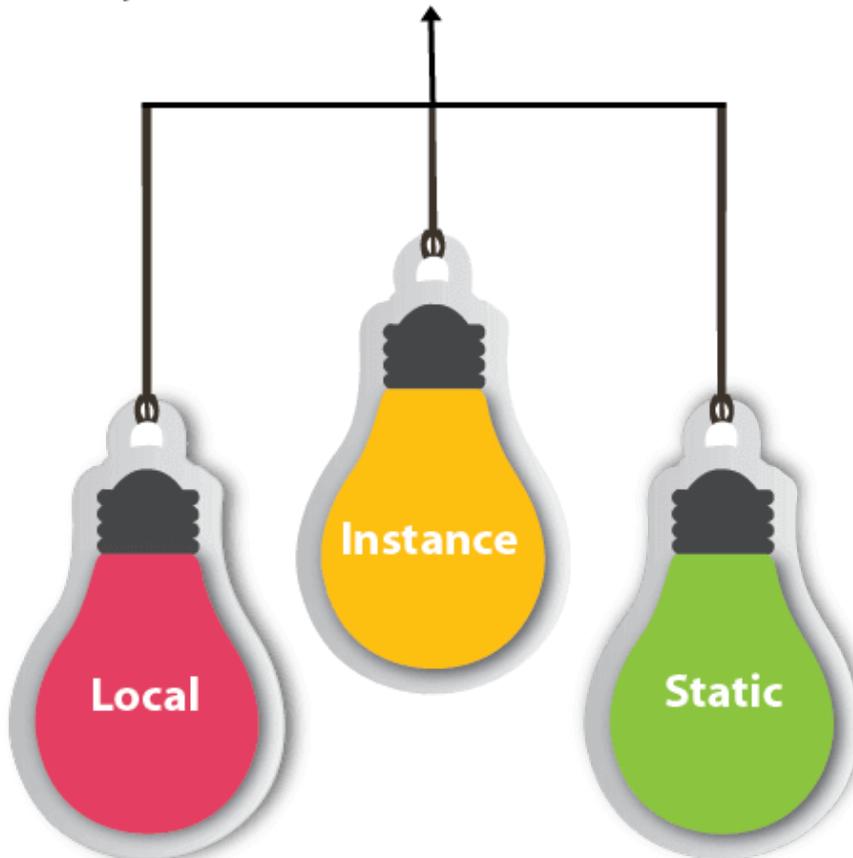


Types of Variables

There are three types of variables in **Java**:

- local variable
- instance variable
- static variable

Types of Variables



1) Local Variable

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

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

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as **static**.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) Static variable

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

Example to understand the types of variables in java

```
public class A
{
    static int m=100;//static variable
    void method()
    {
        int n=90;//local variable
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
    }
}//end of class
```

Java Variable Example: Add Two Numbers

```
public class Simple{
    public static void main(String[] args){
        int a=10;
        int b=10;
        int c=a+b;
        System.out.println(c);
    }
}
```

Output:

```
20
```

Java Variable Example: Widening

```
public class Simple{  
    public static void main(String[] args){  
        int a=10;  
        float f=a;  
        System.out.println(a);  
        System.out.println(f);  
    }  
}
```

Output:

```
10  
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
public class Simple{  
    public static void main(String[] args){  
        float f=10.5f;  
        //int a=f;//Compile time error  
        int a=(int)f;  
        System.out.println(f);  
        System.out.println(a);  
    }  
}
```

Output:

```
10.5  
10
```

Java Variable Example: Overflow

```
class Simple{
```

```
public static void main(String[] args){  
    //Overflow  
    int a=130;  
    byte b=(byte)a;  
    System.out.println(a);  
    System.out.println(b);  
}
```

Output:

```
130  
-126
```

Java Variable Example: Adding Lower Type

```
class Simple{  
    public static void main(String[] args){  
        byte a=10;  
        byte b=10;  
        //byte c=a+b;//Compile Time Error: because a+b=20 will be int  
        byte c=(byte)(a+b);  
        System.out.println(c);  
    }  
}
```

Output:

```
20
```

← Prev

Next →

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes](#), [Interfaces](#), and [Arrays](#).

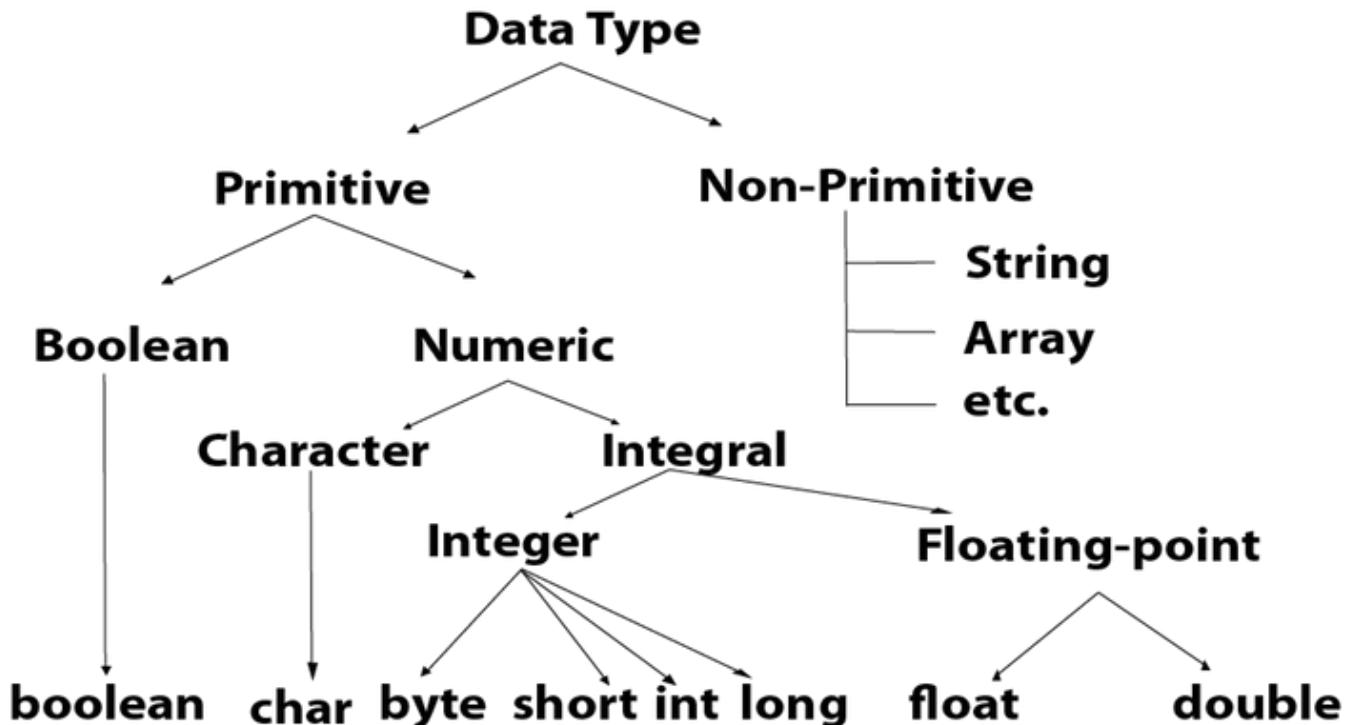
Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in [Java language](#).

Java is a statically-typed programming language. It means, all [variables](#) must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

Boolean one = **false**

Byte Data Type

The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

```
byte a = 10, byte b = -20
```



Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

```
short s = 10000, short r = -5000
```

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (- 2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is -2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

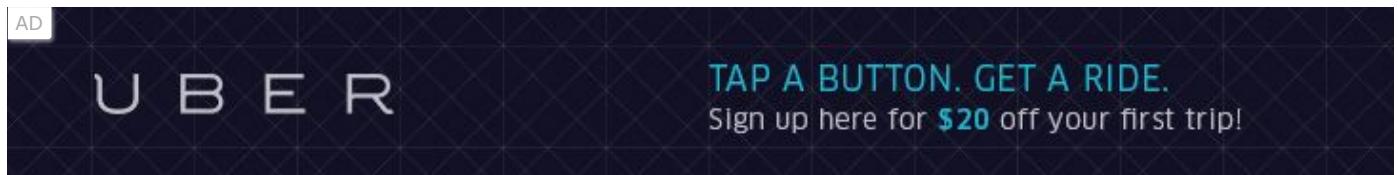
The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

```
int a = 100000, int b = -200000
```

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^{63}) to 9,223,372,036,854,775,807($2^{63} - 1$)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.



Example:

```
long a = 100000L, long b = -200000L
```

Float Data Type

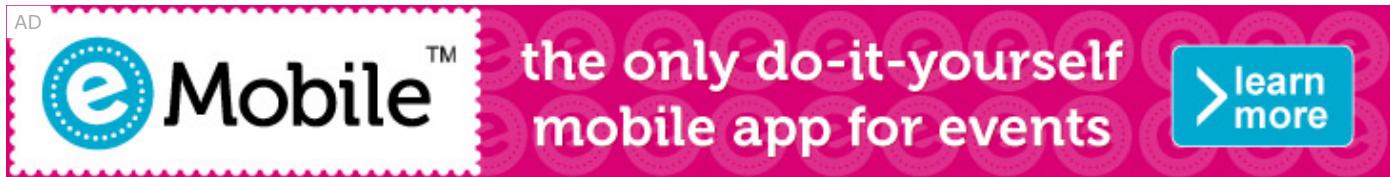
The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

```
float f1 = 234.5f
```

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 12.3
```

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Example:

```
char letterA = 'A'
```

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

[← Prev](#)[Next →](#)

AD

Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for Chinese, and so on.

Problem

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length. Some common characters are encoded as single bytes, others require two or more bytes.

Solution

To solve these problems, a new language standard was developed i.e. Unicode System.

In Unicode, each character holds 2 bytes, so Java also uses 2 bytes for characters.

lowest value: \u0000

highest value: \uFFFF

← Prev

Next →

Operators in Java

Operator in **Java** is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	<i>expr++ expr--</i>
	prefix	<i>++expr --expr +expr -expr ~ !</i>
Arithmetic	multiplicative	<i>* / %</i>
	additive	<i>+ -</i>
Shift	shift	<i><< >> >>></i>
Relational	comparison	<i>< > <= >= instanceof</i>
	equality	<i>== !=</i>
Bitwise	bitwise AND	<i>&</i>
	bitwise exclusive OR	<i>^</i>
	bitwise inclusive OR	<i> </i>

Logical	logical AND	<code>&&</code>
	logical OR	<code> </code>
Ternary	ternary	<code>? :</code>
Assignment	assignment	<code>= += -= *= /= %= &= ^= = <<= >>= >>>=</code>

Java Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- o incrementing/decrementing a value by one
- o negating an expression
- o inverting the value of a boolean

AD

Java Unary Operator Example: ++ and --

```
public class OperatorExample{
    public static void main(String args[]){
        int x=10;
        System.out.println(x++);//10 (11)
        System.out.println(++x);//12
        System.out.println(x--);//12 (11)
        System.out.println(--x);//10
    }
}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example 2: ++ and --

```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=10;  
        System.out.println(a++ + ++a); //10+12=22  
        System.out.println(b++ + b++); //10+11=21  
    }  
}
```

Output:

```
22  
21
```

Java Unary Operator Example: ~ and !

```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=-10;  
        boolean c=true;  
        boolean d=false;  
        System.out.println(~a); // -11 (minus of total positive value which starts from 0)  
        System.out.println(~b); // 9 (positive of total minus, positive starts from 0)  
        System.out.println(!c); // false (opposite of boolean value)  
        System.out.println(!d); // true  
    }  
}
```

Output:

```
-11  
9
```

```
false  
true
```

Java Arithmetic Operators

Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Java Arithmetic Operator Example

```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=5;  
        System.out.println(a+b); //15  
        System.out.println(a-b); //5  
        System.out.println(a*b); //50  
        System.out.println(a/b); //2  
        System.out.println(a%b); //0  
    }  
}
```

Output:

```
15  
5  
50  
2  
0
```

Java Arithmetic Operator Example: Expression

```
public class OperatorExample{  
    public static void main(String args[]){  
        System.out.println(10*10/5+3-1*4/2);  
    }  
}
```

Output:

21

Java Left Shift Operator

The Java left shift operator `<<` is used to shift all of the bits in a value to the left side of a specified number of times.

Java Left Shift Operator Example

```
public class OperatorExample{  
    public static void main(String args[]){  
        System.out.println(10<<2); // 10*2^2 = 10*4 = 40  
        System.out.println(10<<3); // 10*2^3 = 10*8 = 80  
        System.out.println(20<<2); // 20*2^2 = 20*4 = 80  
        System.out.println(15<<4); // 15*2^4 = 15*16 = 240  
    }  
}
```

Output:

```
40  
80  
80  
240
```

Java Right Shift Operator

The Java right shift operator `>>` is used to move the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
public class OperatorExample{  
    public static void main(String args[]){  
        System.out.println(10>>2); // 10/2^2 = 10/4 = 2  
        System.out.println(20>>2); // 20/2^2 = 20/4 = 5  
        System.out.println(20>>3); // 20/2^3 = 20/8 = 2  
    }  
}
```

}}

Output:

```
2  
5  
2
```

Java Shift Operator Example: >> vs >>>

```
public class OperatorExample{  
    public static void main(String args[]){  
        //For positive number, >> and >>> works same  
        System.out.println(20>>2);  
        System.out.println(20>>>2);  
        //For negative number, >>> changes parity bit (MSB) to 0  
        System.out.println(-20>>2);  
        System.out.println(-20>>>2);  
    }  
}
```

Output:

```
5  
5  
-5  
1073741819
```

Java AND Operator Example: Logical && and Bitwise &

The logical `&&` operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise `&` operator always checks both conditions whether first condition is true or false.



```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=5;  
        int c=20;  
        System.out.println(a<b&&a<c);//false && true = false  
        System.out.println(a<b&a<c);//false & true = false  
    }  
}
```

Output:

```
false  
false
```

Java AND Operator Example: Logical && vs Bitwise &

```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=5;  
        int c=20;  
        System.out.println(a<b&&a++<c);//false && true = false  
        System.out.println(a);//10 because second condition is not checked  
        System.out.println(a<b&a++<c);//false && true = false  
        System.out.println(a);//11 because second condition is checked  
    }  
}
```

Output:

```
false  
10  
false  
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=5;  
        int c=20;  
  
        System.out.println(a>b||a<c); //true || true = true  
        System.out.println(a>b|a<c); //true | true = true  
        //|| vs |  
        System.out.println(a>b||a++<c); //true || true = true  
        System.out.println(a); //10 because second condition is not checked  
        System.out.println(a>b|a++<c); //true | true = true  
        System.out.println(a); //11 because second condition is checked  
    }  
}
```

Output:

AD

```
true  
true  
true  
10  
true  
11
```

Java Ternary Operator

Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming. It is the only conditional operator which takes three operands.

Java Ternary Operator Example

```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=2;  
        int b=5;  
        int min=(a<b)?a:b;  
        System.out.println(min);  
    }  
}
```

Output:

```
2
```

Another Example:

```
public class OperatorExample{  
    public static void main(String args[]){  
        int a=10;  
        int b=5;  
        int min=(a<b)?a:b;  
        System.out.println(min);  
    }  
}
```

Output:

```
5
```

Java Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Java Assignment Operator Example

```
public class OperatorExample{
```

```
public static void main(String args[]){
    int a=10;
    int b=20;
    a+=4;//a=a+4 (a=10+4)
    b-=4;//b=b-4 (b=20-4)
    System.out.println(a);
    System.out.println(b);
}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
public class OperatorExample{
    public static void main(String[] args){
        int a=10;
        a+=3;//10+3
        System.out.println(a);
        a-=4;//13-4
        System.out.println(a);
        a*=2;//9*2
        System.out.println(a);
        a/=2;//18/2
        System.out.println(a);
    }
}
```

Output:

```
13
9
18
9
```

Java Assignment Operator Example: Adding short

```
public class OperatorExample{  
    public static void main(String args[]){  
        short a=10;  
        short b=10;  
        //a+=b;//a=a+b internally so fine  
        a=a+b;//Compile time error because 10+10=20 now int  
        System.out.println(a);  
    }  
}
```

Output:

```
Compile time error
```

After type cast:

```
public class OperatorExample{  
    public static void main(String args[]){  
        short a=10;  
        short b=10;  
        a=(short)(a+b);//20 which is int now converted to short  
        System.out.println(a);  
    }  
}
```

Output:

```
20
```

You may also like

[Operator Shifting in Java](#)

← Prev

Next →

Java Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

List of Java Keywords

A list of Java keywords or reserved words are given below:

1. **abstract:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. **boolean:** Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. **break:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. **byte:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. **case:** Java case keyword is used with the switch statements to mark blocks of text.
6. **catch:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. **char:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. **class:** Java class keyword is used to declare a class.
9. **continue:** Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.
10. **default:** Java default keyword is used to specify the default block of code in a switch statement.
11. **do:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. **double:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. **else:** Java else keyword is used to indicate the alternative branches in an if statement.
14. **enum:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. **extends:** Java extends keyword is used to indicate that a class is derived from another class or interface.

16. **final:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. **finally:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. **float:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. **for:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. **if:** Java if keyword tests the condition. It executes the if block if the condition is true.
21. **implements:** Java implements keyword is used to implement an interface.
22. **import:** Java import keyword makes classes and interfaces available and accessible to the current source code.
23. **instanceof:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. **int:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. **interface:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. **long:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. **new:** Java new keyword is used to create new objects.
29. **null:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. **package:** Java package keyword is used to declare a Java package that includes the classes.
31. **private:** Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
32. **protected:** Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. **public:** Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
34. **return:** Java return keyword is used to return from a method when its execution is complete.
35. **short:** Java short keyword is used to declare a variable that can hold a 16-bit integer.

36. **static:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. **super:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. **switch:** The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
40. **synchronized:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. **this:** Java this keyword can be used to refer the current object in a method or constructor.
42. **throw:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. **throws:** The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. **try:** Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. **volatile:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. **while:** Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iteration is not fixed, it is recommended to use the while loop.

← Prev

Next →

Java OOPs Concepts

In this page, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.

The popular object-oriented languages are **Java**, **C#**, **PHP**, **Python**, **C++**, etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

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**
- **Class**
- **Inheritance**
- **Polymorphism**
- **Abstraction**
- **Encapsulation**

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

- **Coupling**
- **Cohesion**
- **Association**
- **Aggregation**
- **Composition**

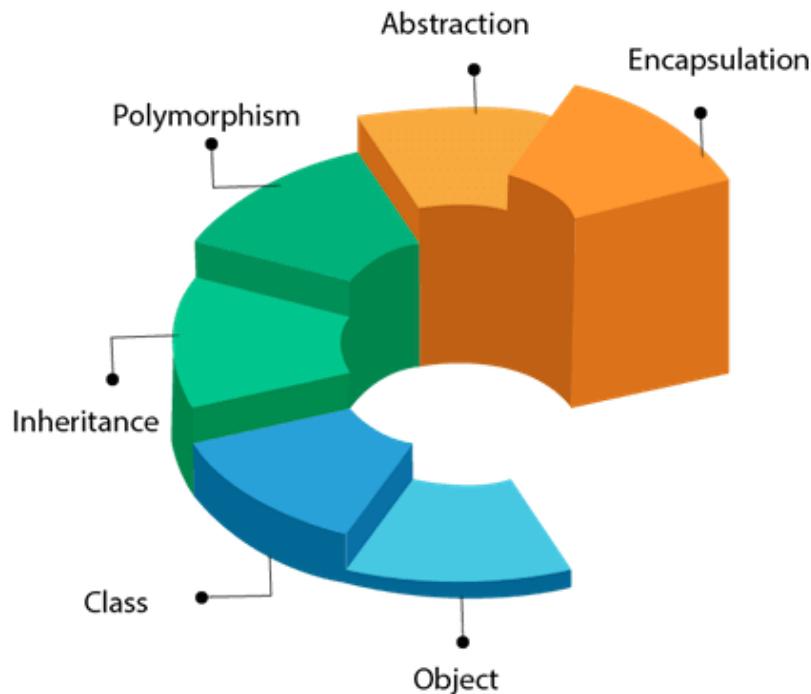
AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

OOPs (Object-Oriented Programming System)



Object



Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response

returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.



Polymorphism

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use **method overloading** and **method overriding** to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

AD



DRAIN CLEANING EQUIPMENT



BREAK THROUGH
BLOCKAGES

FIND YOUR SOLUTION

In Java, we use abstract class and interface to achieve abstraction.



Capsule

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling

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

Cohesion

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

An advertisement for Uber. The top left corner says "AD". The word "UBER" is written in large, bold, white letters. To the right, there is promotional text: "TAP A BUTTON. GET A RIDE." and "Sign up here for \$20 off your first trip!".

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- o One to One
- o One to Many
- o Many to One, and
- o Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

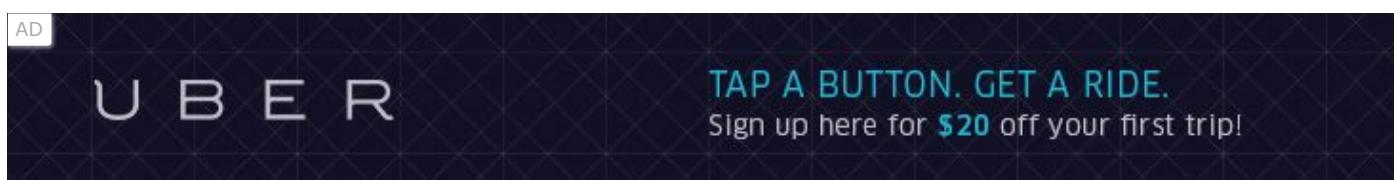
Association can be undirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.



Advantage of OOPs over Procedure-oriented programming language

- 1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.
- 2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

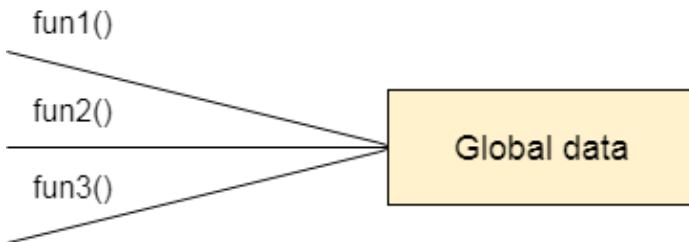


Figure: Data Representation in Procedure-Oriented Programming

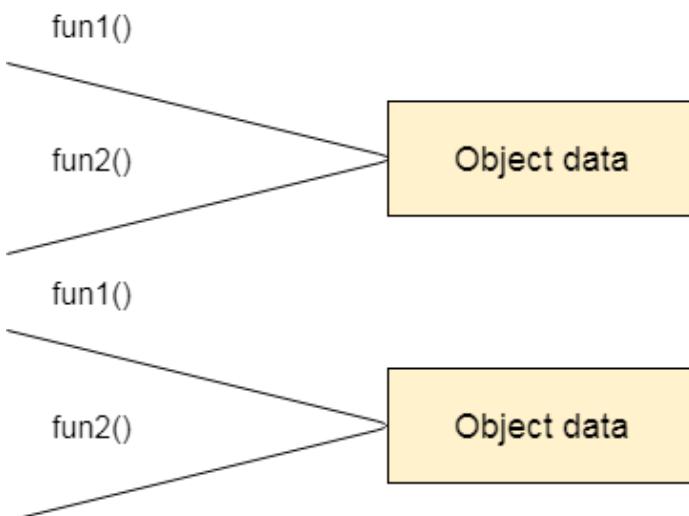


Figure: Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

Do You Know?

- Can we overload the main method?
- A Java Constructor returns a value but, what?
- Can we create a program without main method?
- What are the six ways to use this keyword?
- Why is multiple inheritance not supported in Java?

- Why use aggregation?
- Can we override the static method?
- What is the covariant return type?
- What are the three usages of Java super keyword?
- Why use instance initializer block?
- What is the usage of a blank final variable?
- What is a marker or tagged interface?
- What is runtime polymorphism or dynamic method dispatch?
- What is the difference between static and dynamic binding?
- How downcasting is possible in Java?
- What is the purpose of a private constructor?
- What is object cloning?

What will we learn in OOPs Concepts?

- Advantage of OOPs
- Naming Convention
- Object and class
- Method overloading
- Constructor
- static keyword
- this keyword with six usage
- Inheritance
- Aggregation
- Method Overriding
- Covariant Return Type
- super keyword
- Instance Initializer block
- final keyword
- Abstract class
- Interface
- Runtime Polymorphism

- o Static and Dynamic Binding
- o Downcasting with instanceof operator
- o Package
- o Access Modifiers
- o Encapsulation
- o Object Cloning

[← Prev](#)[Next →](#)

AD



For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- o Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

Java Naming Convention

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of Naming Conventions in Java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

Naming Conventions of the Different Identifiers

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

Identifiers Type	Naming Rules	Examples
Class	<p>It should start with the uppercase letter.</p> <p>It should be a noun such as Color, Button, System, Thread, etc.</p> <p>Use appropriate words, instead of acronyms.</p>	<pre>public class Employee { //code snippet }</pre>
Interface	<p>It should start with the uppercase letter.</p> <p>It should be an adjective such as Runnable, Remote, ActionListener.</p> <p>Use appropriate words, instead of acronyms.</p>	<pre>interface Printable { //code snippet }</pre>

Method	<p>It should start with lowercase letter.</p> <p>It should be a verb such as main(), print(), println().</p> <p>If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().</p>	<pre>class Employee { // method void draw() { //code snippet } }</pre>
Variable	<p>It should start with a lowercase letter such as id, name.</p> <p>It should not start with the special characters like & (ampersand), \$ (dollar), _ (underscore).</p> <p>If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.</p> <p>Avoid using one-character variables such as x, y, z.</p>	<pre>class Employee { // variable int id; //code snippet }</pre>
Package	<p>It should be a lowercase letter such as java, lang.</p> <p>If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.</p>	<pre>//package package com.javatpoint; class Employee { //code snippet }</pre>
Constant	<p>It should be in uppercase letters such as RED, YELLOW.</p> <p>If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.</p> <p>It may contain digits but not as the first letter.</p>	<pre>class Employee { //constant static final int MIN AGE = 18; //code snippet }</pre>

CamelCase in Java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Objects and Classes in Java

In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

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

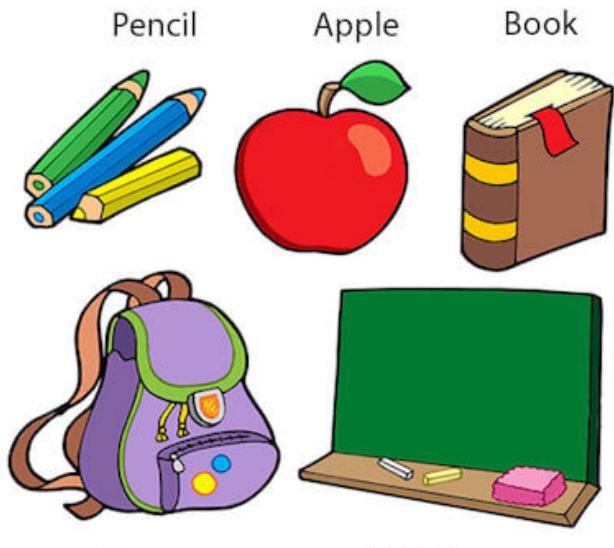
What is an object in Java

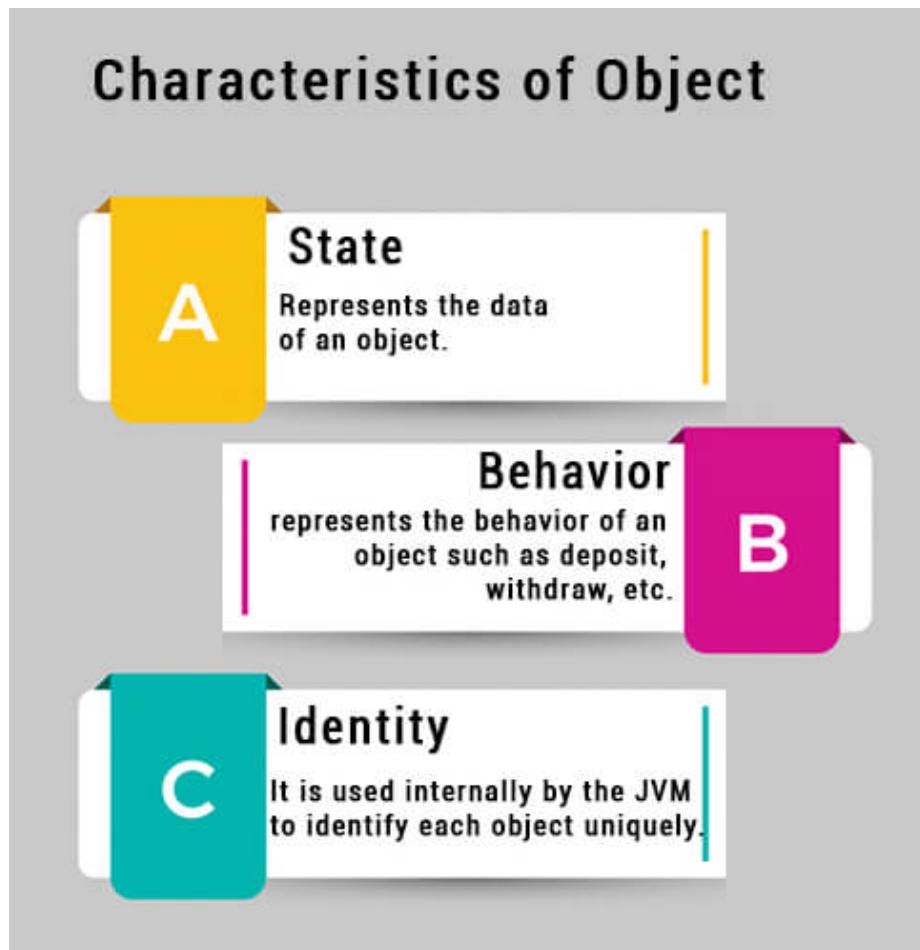
An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

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.

Objects: Real World Examples





For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

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.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

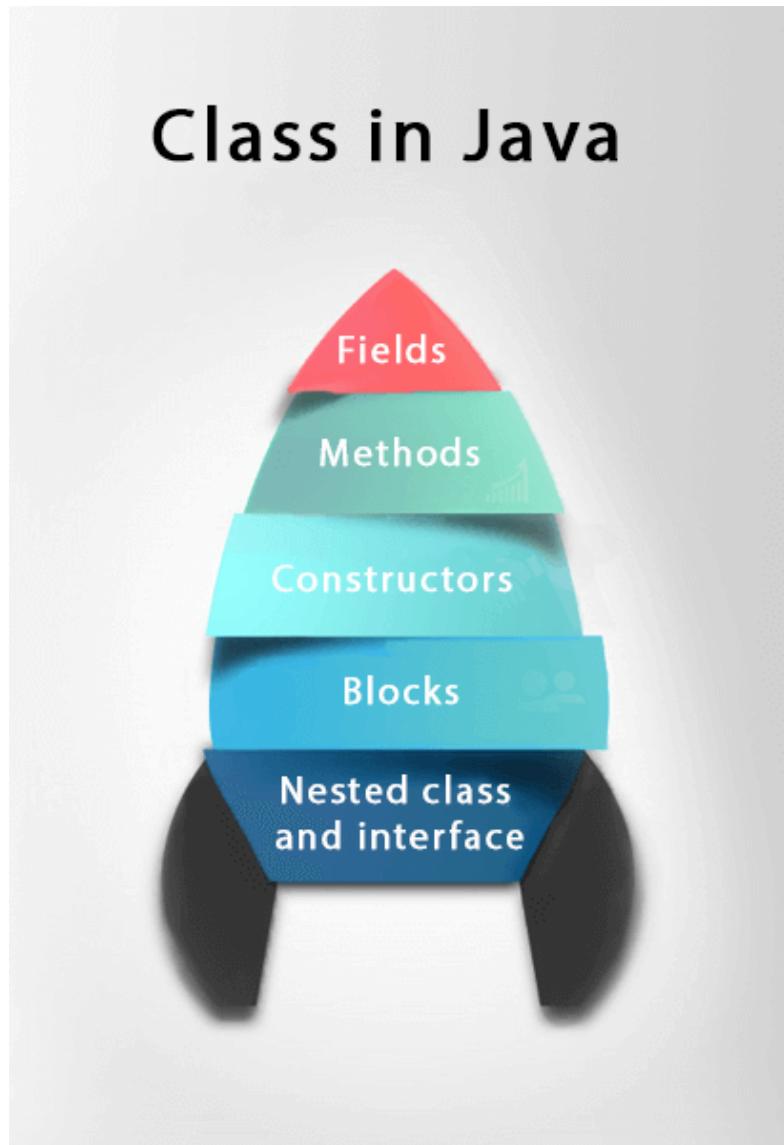


What is a class in Java

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**



Syntax to declare a class:

```
class <class_name>{  
    field;  
    method;  
}
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

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: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.

class Student{
    //defining fields
    int id;//field or data member or instance variable
    String name;
    //creating main method inside the Student class
    public static void main(String args[]){
        //Creating an object or instance
```

```

Student s1=new Student();//creating an object of Student
//Printing values of the object
System.out.println(s1.id);//accessing member through reference variable
System.out.println(s1.name);
}
}

```

Test it Now

Output:

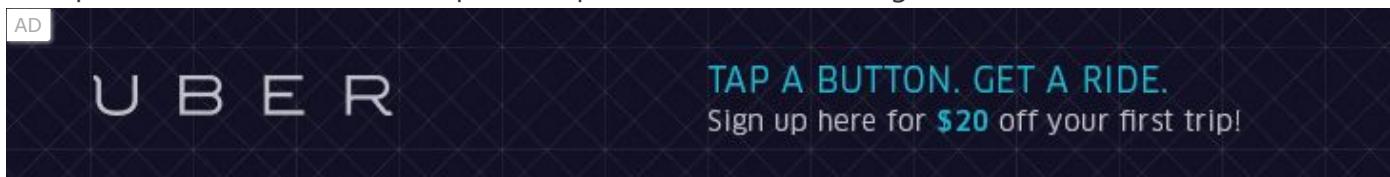
```

0
null

```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.



We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

```

//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
    int id;
    String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
    public static void main(String args[]){
        Student s1=new Student();

```

```

System.out.println(s1.id);
System.out.println(s1.name);
}
}

```

Test it Now

Output:

```

0
null

```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

File: TestStudent2.java

```

class Student{
    int id;
    String name;
}

class TestStudent2{
    public static void main(String args[]){
        Student s1=new Student();
        s1.id=101;
        s1.name="Sonoo";
    }
}

```

```
System.out.println(s1.id+" "+s1.name);//printing members with a white space  
}  
}
```

Test it Now

Output:

```
101 Sonoo
```

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

```
class Student{  
    int id;  
    String name;  
}  
  
class TestStudent3{  
    public static void main(String args[]){  
        //Creating objects  
        Student s1=new Student();  
        Student s2=new Student();  
        //Initializing objects  
        s1.id=101;  
        s1.name="Sonoo";  
        s2.id=102;  
        s2.name="Amit";  
        //Printing data  
        System.out.println(s1.id+" "+s1.name);  
        System.out.println(s2.id+" "+s2.name);  
    }  
}
```

Test it Now

Output:

101 Sonoo

102 Amit

2) 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

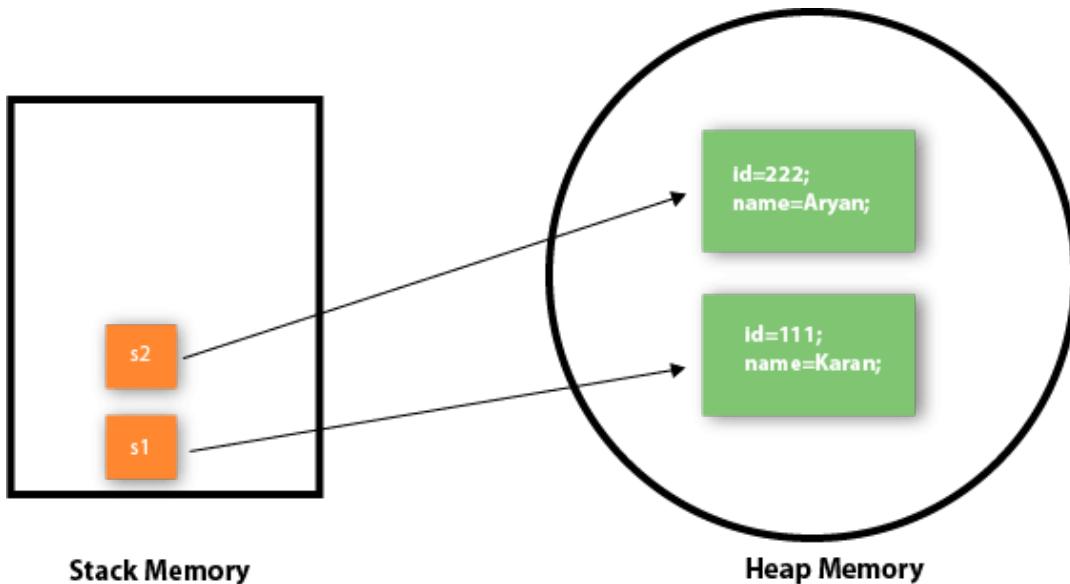
```
class Student{  
    int rollno;  
    String name;  
    void insertRecord(int r, String n){  
        rollno=r;  
        name=n;  
    }  
    void displayInformation(){System.out.println(rollno+" "+name);}  
}  
  
class TestStudent4{  
    public static void main(String args[]){  
        Student s1=new Student();  
        Student s2=new Student();  
        s1.insertRecord(111,"Karan");  
        s2.insertRecord(222,"Aryan");  
        s1.displayInformation();  
        s2.displayInformation();  
    }  
}
```

Test it Now

Output:

111 Karan

222 Aryan



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

public class TestEmployee {
    public static void main(String[] args) {
```

```

Employee e1=new Employee();
Employee e2=new Employee();
Employee e3=new Employee();
e1.insert(101,"ajeet",45000);
e2.insert(102,"irfan",25000);
e3.insert(103,"nakul",55000);
e1.display();
e2.display();
e3.display();
}
}

```

Test it Now

Output:

```

101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0

```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```

class Rectangle{
    int length;
    int width;
    void insert(int l, int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}

class TestRectangle1{
    public static void main(String args[]){
        Rectangle r1=new Rectangle();
        Rectangle r2=new Rectangle();
    }
}

```

```
r1.insert(11,5);
r2.insert(3,15);
r1.calculateArea();
r2.calculateArea();

}
```

Test it Now

Output:

```
55
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.

Different ways to create an object in Java

- 1 By new keyword
- 2 By `newInstance()` method
- 3 By `clone()` method
- 4 By deserialization
- 5 By factory method etc.



AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
new Calculation() //anonymous object
```

Calling method through a reference:

```
Calculation c=new Calculation();
c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
class Calculation{
    void fact(int n){
        int fact=1;
        for(int i=1;i<=n;i++){
            fact=fact*i;
        }
        System.out.println("factorial is "+fact);
    }
    public static void main(String args[]){
        new Calculation().fact(5);//calling method with anonymous object
    }
}
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
int a=10, b=20;
```

Initialization of reference variables:

```
Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```
//Java Program to illustrate the use of Rectangle class which
//has length and width data members

class Rectangle{
    int length;
    int width;
    void insert(int l,int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}

class TestRectangle2{
    public static void main(String args[]){
        Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
        r1.insert(11,5);
        r2.insert(3,15);
        r1.calculateArea();
        r2.calculateArea();
    }
}
```

Test it Now

Output:

```
55
45
```

Real World Example: Account

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods

class Account{
    int acc_no;
    String name;
```

```
float amount;  
//Method to initialize object  
void insert(int a, String n, float amt){  
    acc_no=a;  
    name=n;  
    amount=amt;  
}  
//deposit method  
void deposit(float amt){  
    amount=amount+amt;  
    System.out.println(amt+ " deposited");  
}  
//withdraw method  
void withdraw(float amt){  
    if(amount<amt){  
        System.out.println("Insufficient Balance");  
    }  
    else{  
        amount=amount-amt;  
        System.out.println(amt+ " withdrawn");  
    }  
}  
//method to check the balance of the account  
void checkBalance(){System.out.println("Balance is: "+amount);}  
//method to display the values of an object  
void display(){System.out.println(acc_no+ " "+name+ " "+amount);}  
}  
//Creating a test class to deposit and withdraw amount  
class TestAccount{  
    public static void main(String[] args){  
        Account a1=new Account();  
        a1.insert(832345, "Ankit", 1000);  
        a1.display();  
        a1.checkBalance();  
        a1.deposit(40000);  
        a1.checkBalance();  
        a1.withdraw(15000);  
        a1.checkBalance();  
    }  
}
```

Test it Now**Output:**

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our YouTube Channel: Join Now](#)**Feedback**

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Constructors in Java

In **Java**, a constructor is a block of codes similar to the method. It is called when an instance of the **class** is created. At the time of calling constructor, memory for the object is allocated in the memory.

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

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

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

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. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

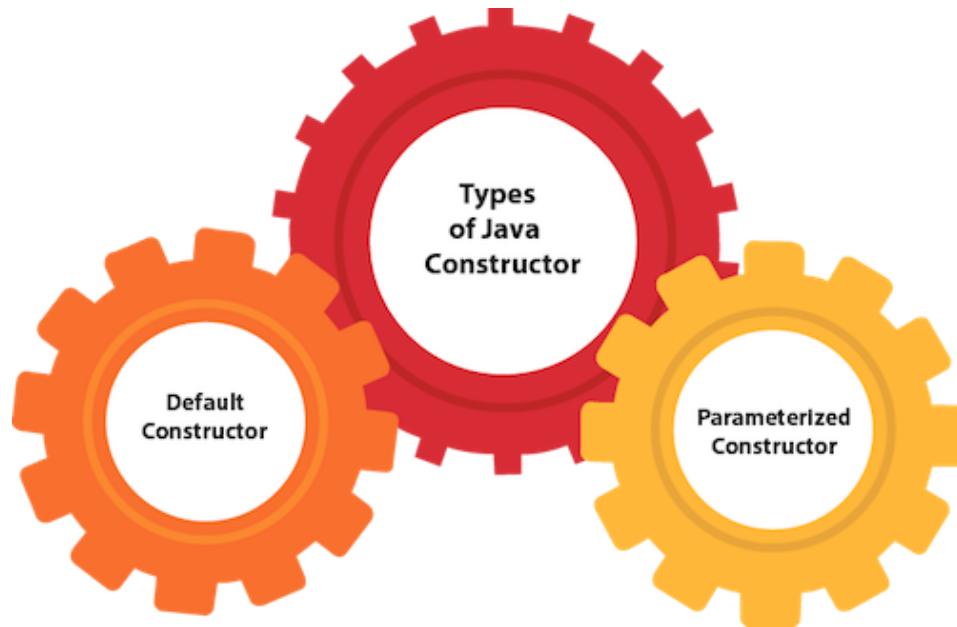
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use **access modifiers** while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

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.

Syntax of default constructor:

```
<class_name>(){}
```

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.

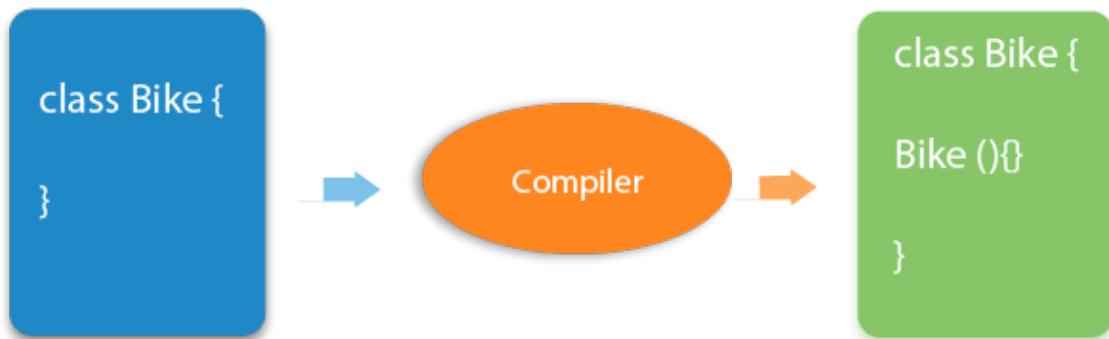
```
//Java Program to create and call a default constructor
class Bike1{
    //creating a default constructor
    Bike1(){System.out.println("Bike is created");}
    //main method
    public static void main(String args[]){
        //calling a default constructor
        Bike1 b=new Bike1();
    }
}
```

[Test it Now](#)

Output:

```
Bike is created
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



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

```

//Let us see another example of default constructor
//which displays the default values
class Student3{
    int id;
    String name;
    //method to display the value of id and name
    void display(){System.out.println(id+ " "+name);}

    public static void main(String args[]){
        //creating objects
        Student3 s1=new Student3();
        Student3 s2=new Student3();
        //displaying values of the object
        s1.display();
        s2.display();
    }
}
  
```

```
}
```

```
}
```

Test it Now

Output:

```
0 null  
0 null
```

Explanation: In the above class, you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

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. However, you can provide the same values also.

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.
```

```
class Student4{  
    int id;  
    String name;  
    //creating a parameterized constructor  
    Student4(int i, String n){  
        id = i;  
        name = n;  
    }  
    //method to display the values  
    void display(){System.out.println(id + " " + name);}  
}
```

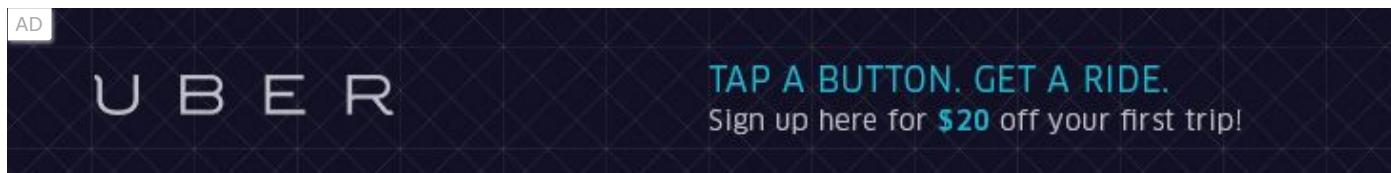
```
public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
}
```

Test it Now

Output:



111 Karan
222 Aryan



Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor **overloading in Java** is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
```

```

int age;
//creating two arg constructor
Student5(int i,String n){
id = i;
name = n;
}
//creating three arg constructor
Student5(int i,String n,int a){
id = i;
name = n;
age=a;
}
void display(){System.out.println(id+ " "+name+ " "+age);}

```

```

public static void main(String args[]){
Student5 s1 = new Student5(111,"Karan");
Student5 s2 = new Student5(222,"Aryan",25);
s1.display();
s2.display();
}
}

```

Test it Now

Output:

```

111 Karan 0
222 Aryan 25

```

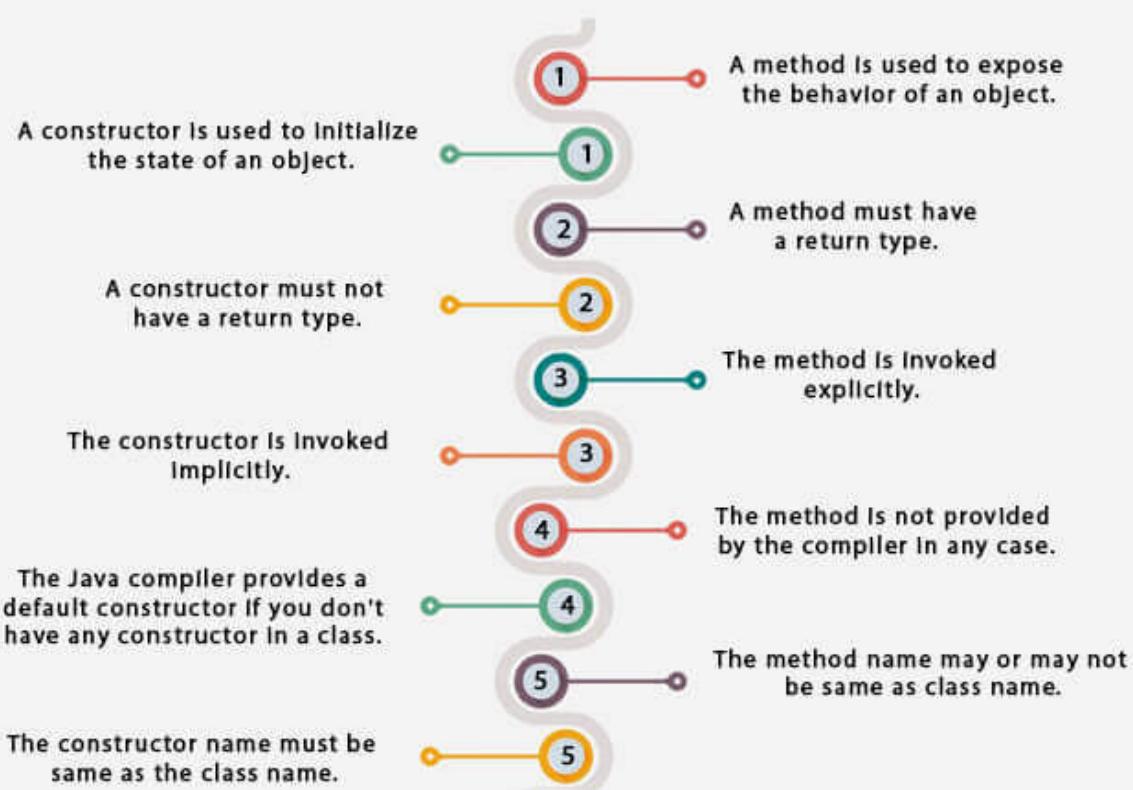
Difference between constructor and method in Java

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

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.

A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Difference between constructor and method in Java



Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

AD

RIDGID
DRAIN CLEANING EQUIPMENT

**BREAK THROUGH
BLOCKAGES**

FIND YOUR SOLUTION

There are many ways to copy the values of one object into another in Java. They are:

- o By constructor
- o By assigning the values of one object into another
- o By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
//Java program to initialize the values from one object to another object.
```

```
class Student6{  
    int id;  
    String name;  
    //constructor to initialize integer and string  
    Student6(int i,String n){  
        id = i;  
        name = n;  
    }  
    //constructor to initialize another object  
    Student6(Student6 s){  
        id = s.id;  
        name = s.name;  
    }  
    void display(){System.out.println(id+" "+name);}  
  
    public static void main(String args[]){  
        Student6 s1 = new Student6(111,"Karan");  
        Student6 s2 = new Student6(s1);  
        s1.display();  
        s2.display();  
    }  
}
```

Test it Now

Output:

```
111 Karan  
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Student7{  
    int id;  
    String name;  
    Student7(int i, String n){  
        id = i;  
        name = n;  
    }  
    Student7(){  
    }  
    void display(){System.out.println(id + " " + name);}  
  
    public static void main(String args[]){  
        Student7 s1 = new Student7(111, "Karan");  
        Student7 s2 = new Student7();  
        s2.id=s1.id;  
        s2.name=s1.name;  
        s1.display();  
        s2.display();  
    }  
}
```

Test it Now

Output:

```
111 Karan  
111 Karan
```

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the `java.lang.reflect` package.

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

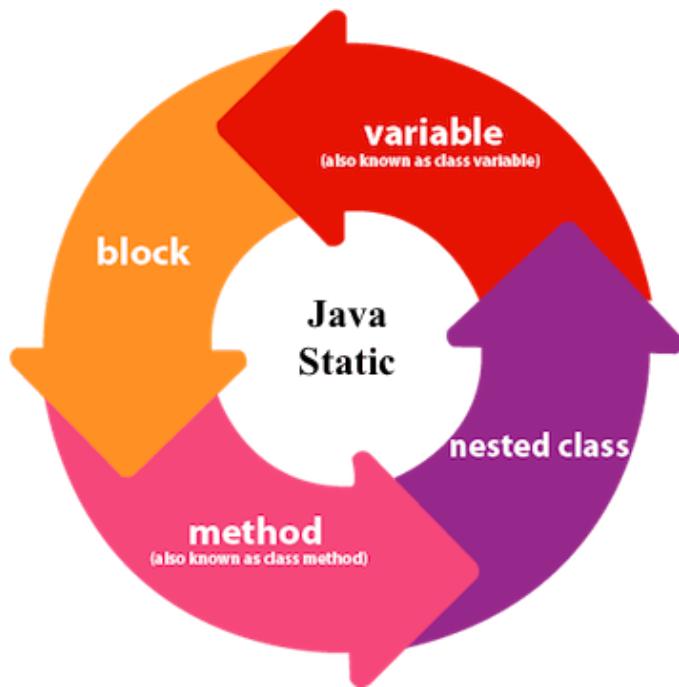


Java static keyword

The **static keyword** in **Java** is used for memory management mainly. We can apply static keyword with **variables**, methods, blocks and **nested classes**. 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
4. Nested class



1) Java static variable

If you declare any variable as static, it is known as a 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.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{  
    int rollno;  
    String name;  
    String college="ITS";  
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all **objects**. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

Example of static variable

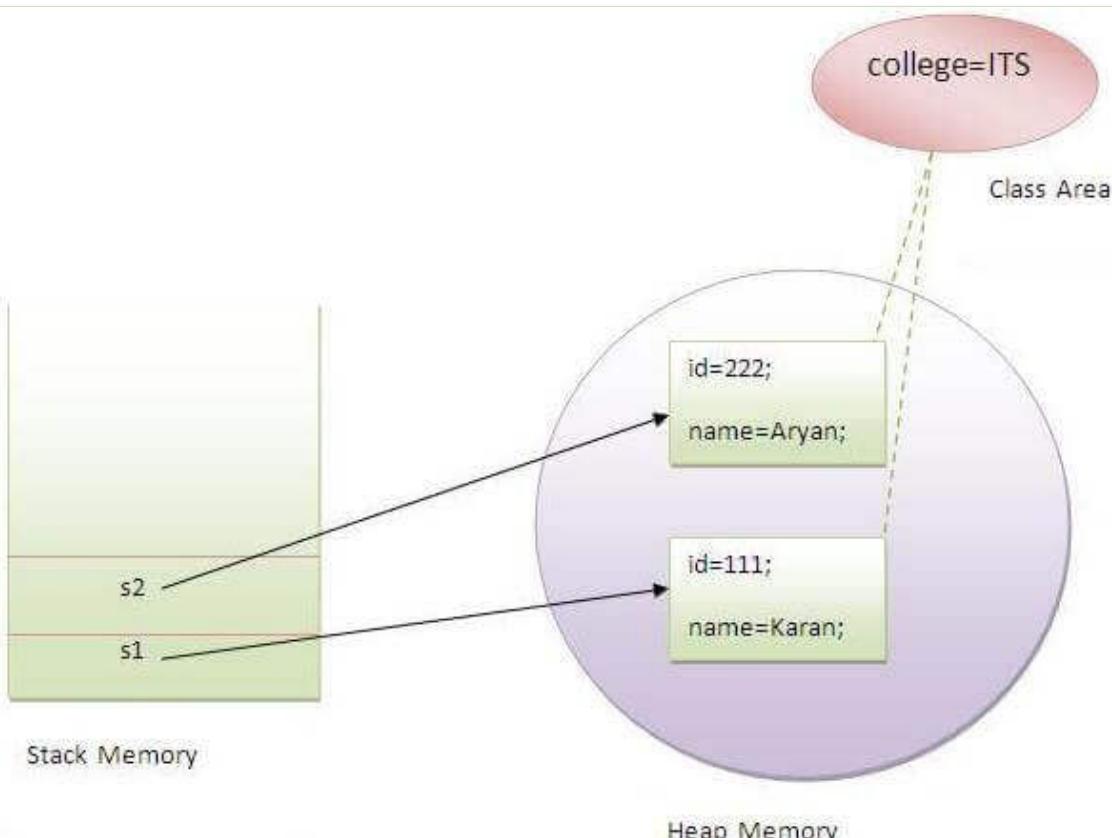
```
//Java Program to demonstrate the use of static variable  
class Student{  
    int rollno;//instance variable  
    String name;  
    static String college ="ITS";//static variable  
    //constructor  
    Student(int r, String n){  
        rollno = r;  
        name = n;  
    }  
    //method to display the values  
    void display (){System.out.println(rollno+ " "+name+ " "+college);}  
}  
  
//Test class to show the values of objects  
public class TestStaticVariable1{  
    public static void main(String args[]){  
        Student s1 = new Student(111,"Karan");  
        Student s2 = new Student(222,"Aryan");  
        //we can change the college of all objects by the single line of code
```

```
//Student.college="BBDIT";
s1.display();
s2.display();
}
```

Test it Now

Output:

```
111 Karan ITS
222 Aryan ITS
```



Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.
```

```
class Counter{
    int count=0;//will get memory each time when the instance is created

    Counter(){
        count++;//incrementing value
        System.out.println(count);
    }

    public static void main(String args[]){
        //Creating objects
        Counter c1=new Counter();
        Counter c2=new Counter();
        Counter c3=new Counter();
    }
}
```

Test it Now

Output:

```
1
1
1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.

class Counter2{
    static int count=0;//will get memory only once and retain its value

    Counter2(){
        count++;//incrementing the value of static variable
        System.out.println(count);
    }
}
```

```
public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Test it Now

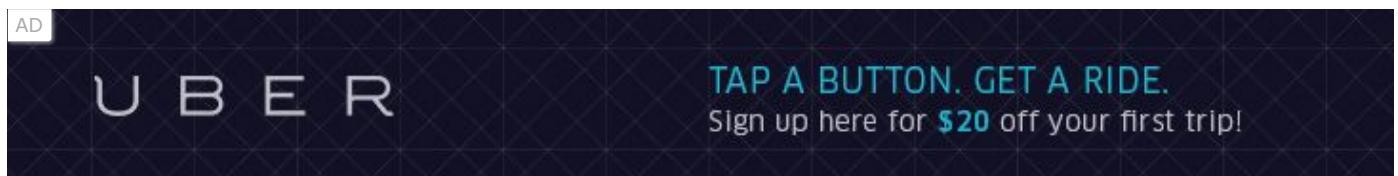
Output:

```
1
2
3
```

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

```
//Java Program to demonstrate the use of a static method.

class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
```

```

college = "BBDIT";
}

//constructor to initialize the variable
Student(int r, String n){
rollno = r;
name = n;
}

//method to display values
void display(){System.out.println(rollno+" "+name+" "+college);}

}

//Test class to create and display the values of object
public class TestStaticMethod{

public static void main(String args[]){
Student.change(); //calling change method
//creating objects
Student s1 = new Student(111,"Karan");
Student s2 = new Student(222,"Aryan");
Student s3 = new Student(333,"Sonoo");
//calling display method
s1.display();
s2.display();
s3.display();
}
}

```

Test it Now

Output:111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT

Another example of a static method that performs a normal calculation

```

//Java Program to get the cube of a given number using the static method

class Calculate{
static int cube(int x){

```

```
return x*x*x;  
}  
  
public static void main(String args[]){  
    int result=Calculate.cube(5);  
    System.out.println(result);  
}  
}
```

Test it Now

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.

```
class A{  
    int a=40;//non static  
  
public static void main(String args[]){  
    System.out.println(a);  
}
```

Test it Now

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 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

```
class A2{
    static{System.out.println("static block is invoked");}
    public static void main(String args[]){
        System.out.println("Hello main");
    }
}
```

Test it Now

Output:static block is invoked
Hello main

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the **main** method.

```
class A3{
    static{
        System.out.println("static block is invoked");
        System.exit(0);
    }
}
```

Test it Now

Output:

static block is invoked

Since JDK 1.7 and above, output would be:

```
Error: Main method not found in class A3, please define the main method as:  
public static void main(String[] args)  
or a JavaFX application class must extend javafx.application.Application
```

[← Prev](#)[Next →](#)

AD

 [Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

[!\[\]\(ec505db02b43f2ccd092e8a2f76029b0_img.jpg\) Splunk tutorial](#)

Splunk

[!\[\]\(5068c970df2bc94a5ebbea3615f6c0ed_img.jpg\) SPSS tutorial](#)

SPSS

[!\[\]\(03a7d1c05e4d566925f90034287f2684_img.jpg\) Swagger tutorial](#)

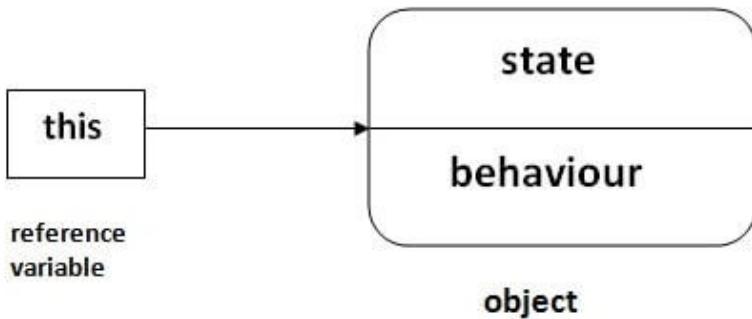
Swagger

[!\[\]\(07d9ae3511207e1c031d0fd89f208c10_img.jpg\) T-SQL tutorial](#)

Transact-SQL

this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

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

Usage of Java this Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

01

this can be used to refer current class instance variable.

04

this can be passed as an argument in the method call.

02

this can be used to invoke current class method (implicity)

05

this can be passed as argument in the constructor call.

03

this() can be used to invoke current class Constructor.

06

this can be used to return the current class instance from the method

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.

Understanding the problem without this keyword

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

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        rollno=rollno;
        name=name;
        fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
    }
}
```

```
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Test it Now

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

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Test it Now

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

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int r, String n, float f){
        rollno=r;
        name=n;
        fee=f;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis3{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }
}
```

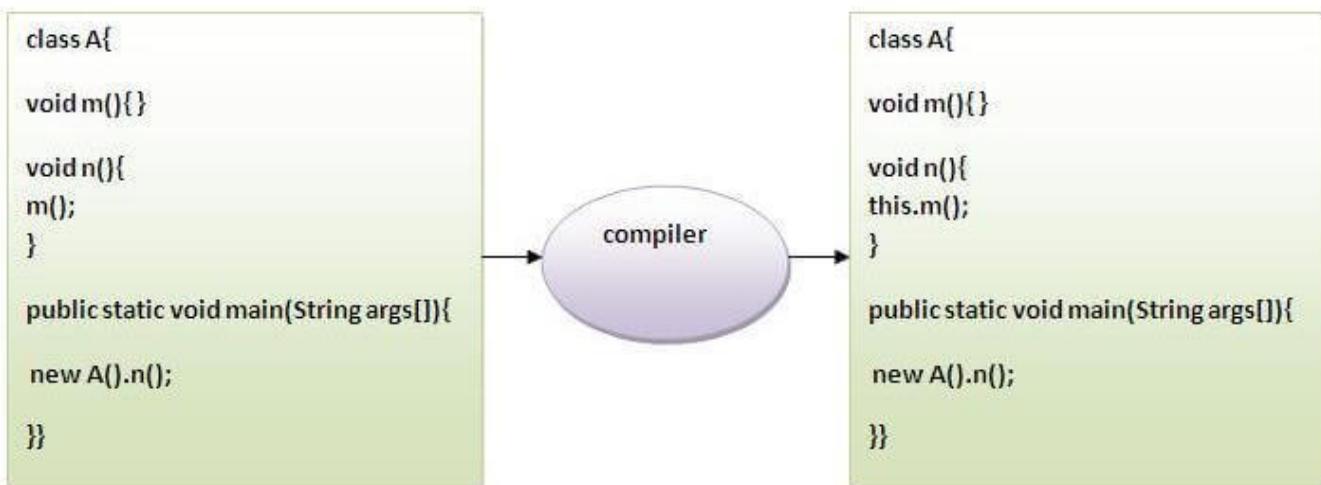
Test it Now**Output:**

```
111 ankit 5000.0
112 sumit 6000.0
```

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

2) this: to invoke current class method

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



```

class A{
    void m(){System.out.println("hello m");}
    void n(){
        System.out.println("hello n");
        //m();//same as this.m()
        this.m();
    }
}

```

```

class TestThis4{
    public static void main(String args[]){
        A a=new A();
        a.n();
    }
}

```

Test it Now

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:

```
class A{  
    A(){System.out.println("hello a");}  
    A(int x){  
        this();  
        System.out.println(x);  
    }  
}  
  
class TestThis5{  
    public static void main(String args[]){  
        A a=new A(10);  
    }  
}
```

Test it Now

Output:

```
hello a  
10
```

Calling parameterized constructor from default constructor:

```
class A{  
    A(){  
        this(5);  
        System.out.println("hello a");  
    }  
    A(int x){
```

```

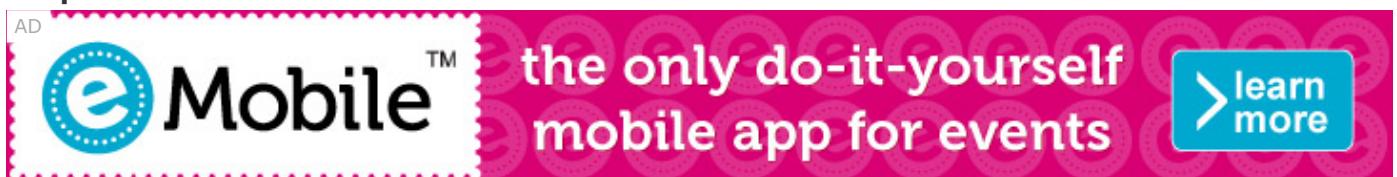
System.out.println(x);
}
}

class TestThis6{
public static void main(String args[]){
A a=new A();
}}

```

[Test it Now](#)

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.

```

class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}

```

```

}

class TestThis7{

public static void main(String args[]){
    Student s1=new Student(111,"ankit","java");
    Student s2=new Student(112,"sumit","java",6000f);
    s1.display();
    s2.display();
}
}

```

Test it Now

Output:

```

111 ankit java 0.0
112 sumit java 6000.0

```

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

```

class Student{
    int rollno;
    String name, course;
    float fee;
    Student(int rollno, String name, String course){
        this.rollno=rollno;
        this.name=name;
        this.course=course;
    }
    Student(int rollno, String name, String course, float fee){
        this.fee=fee;
        this(rollno, name, course); //C.T.Error
    }
    void display(){System.out.println(rollno + " " + name + " " + course + " " + fee);}
}
class TestThis8{

public static void main(String args[]){
    Student s1=new Student(111,"ankit","java");
}

```

```
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Test it Now

Output:

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

4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
    void m(S2 obj){
        System.out.println("method is invoked");
    }
    void p(){
        m(this);
    }
    public static void main(String args[]){
        S2 s1 = new S2();
        s1.p();
    }
}
```

Test it Now

Output:



method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{  
    A4 obj;  
    B(A4 obj){  
        this.obj=obj;  
    }  
    void display(){  
        System.out.println(obj.data);//using data member of A4 class  
    }  
}  
  
class A4{  
    int data=10;  
    A4(){  
        B b=new B(this);  
        b.display();  
    }  
    public static void main(String args[]){  
        A4 a=new A4();  
    }  
}
```

Test it Now

Output:10

6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
return_type method_name(){  
    return this;  
}
```

Example of this keyword that you return as a statement from the method

```
class A{  
    A getA(){  
        return this;  
    }  
    void msg(){System.out.println("Hello java");}  
}  
  
class Test1{  
    public static void main(String args[]){  
        new A().getA().msg();  
    }  
}
```

Test it Now

Output:

```
Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
class A5{
```

```
void m(){  
    System.out.println(this); //prints same reference ID  
}  
  
public static void main(String args[]){  
    A5 obj=new A5();  
    System.out.println(obj); //prints the reference ID  
    obj.m();  
}  
}
```

Test it Now

Output:

```
A5@22b3ea59  
A5@22b3ea59
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

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.

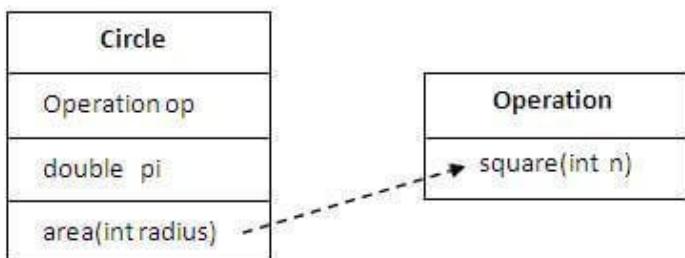
```
class Employee{
    int id;
    String name;
    Address address; //Address is a class
    ...
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- o For Code Reusability.

Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
class Operation{
    int square(int n){
        return n*n;
    }
}
```

```

class Circle{
    Operation op;//aggregation
    double pi=3.14;

    double area(int radius){
        op=new Operation();
        int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
        return pi*rsquare;
    }

    public static void main(String args[]){
        Circle c=new Circle();
        double result=c.area(5);
        System.out.println(result);
    }
}

```

Test it Now

Output:78.5

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.



Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```
public class Address {  
    String city,state,country;  
  
    public Address(String city, String state, String country) {  
        this.city = city;  
        this.state = state;  
        this.country = country;  
    }  
}
```

Emp.java

```
public class Emp {  
    int id;  
    String name;  
    Address address;  
  
    public Emp(int id, String name,Address address) {  
        this.id = id;  
        this.name = name;  
        this.address=address;  
    }  
  
    void display(){  
        System.out.println(id+" "+name);  
        System.out.println(address.city+" "+address.state+" "+address.country);  
    }  
  
    public static void main(String[] args) {  
        Address address1=new Address("gzb","UP","india");  
        Address address2=new Address("gno","UP","india");  
  
        Emp e=new Emp(111,"varun",address1);  
    }  
}
```

```
Emp e2=new Emp(112,"arun",address2);

e.display();
e2.display();

}

}
```

Test it Now

```
Output:111 varun
      gzb UP india
      112 arun
      gno UP india
```

[download this example](#)

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

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** (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new **classes** that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

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

Why use inheritance in java

- For **Method Overriding** (so **runtime polymorphism** can be achieved).
- For **Code Reusability**.

Terms used in Inheritance

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



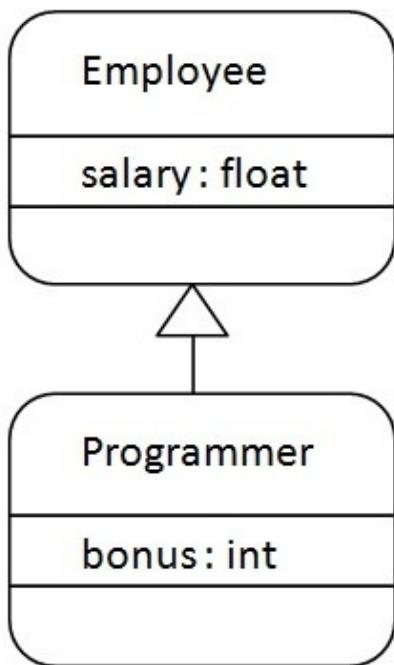
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

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

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

Java Inheritance Example



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

```

class Employee{
    float salary=40000;
}

class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
  
```

Test it Now

```
Programmer salary is:40000.0
```

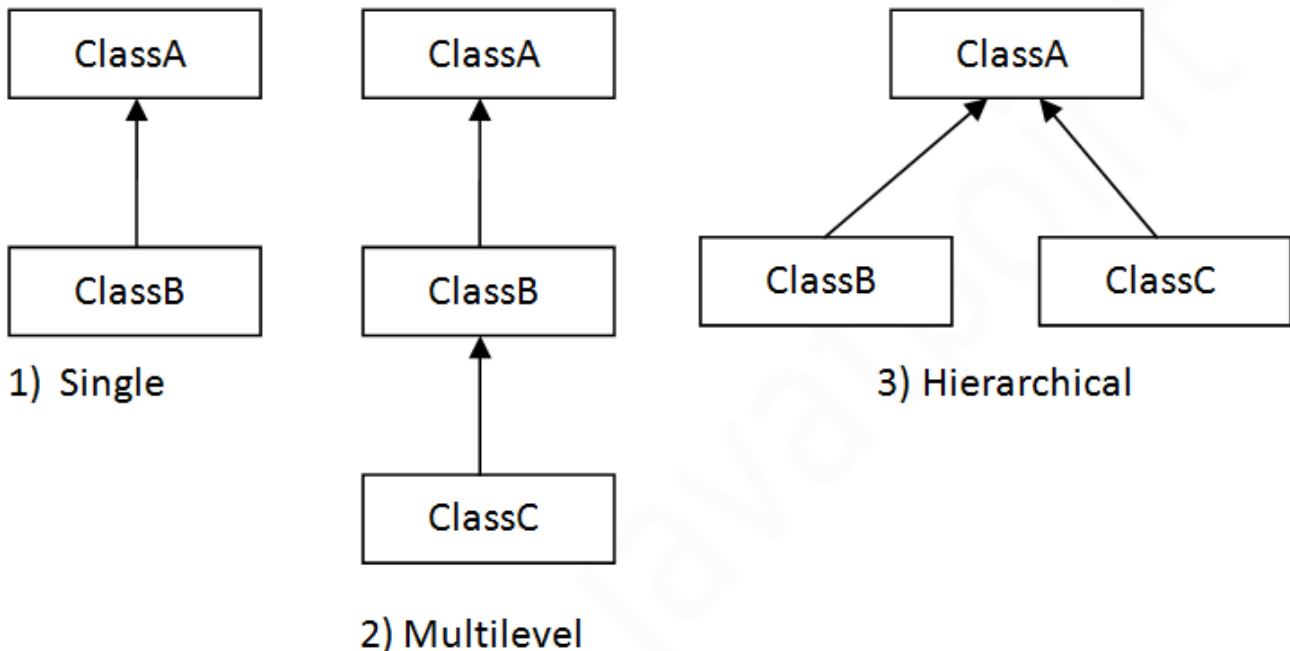
```
Bonus of programmer is:10000
```

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

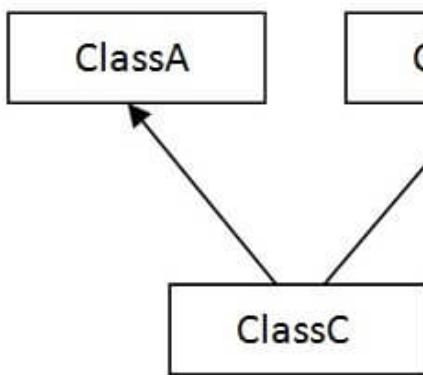
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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

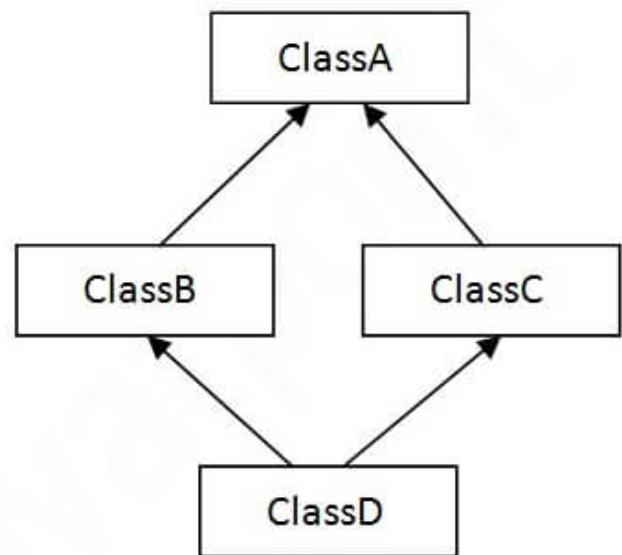


Note: Multiple inheritance is not supported in Java through class.

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



4) Multiple



5) Hybrid

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

```

class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}

class TestInheritance{
    public static void main(String args[]){
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}
  
```

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

```
class Animal{
    void eat(){System.out.println("eating...");}
}

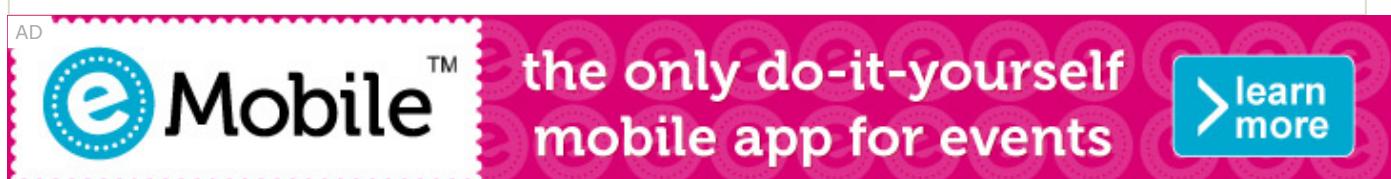
class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}

class BabyDog extends Dog{
    void weep(){System.out.println("weeping...");}
}

class TestInheritance2{
    public static void main(String args[]){
        BabyDog d=new BabyDog();
        d.weep();
        d.bark();
        d.eat();
    }
}
```

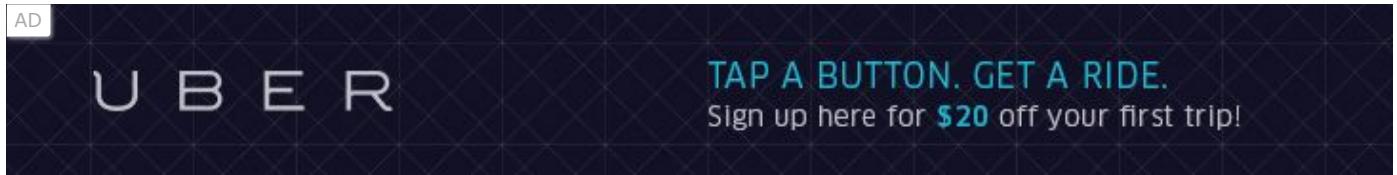
Output:

```
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

```

class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}

class Cat extends Animal{
    void meow(){System.out.println("meowing...");}
}

class TestInheritance3{
    public static void main(String args[]){
        Cat c=new Cat();
        c.meow();
        c.eat();
        //c.bark();//C.T.Error
    }
}

```

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.



```
class A{
    void msg(){System.out.println("Hello");}
}

class B{
    void msg(){System.out.println("Welcome");}
}

class C extends A,B{//suppose if it were

public static void main(String args[]){
    C obj=new C();
    obj.msg(); //Now which msg() method would be invoked?
}
}
```

Test it Now

Compile Time Error

← Prev

Next →

Method Overloading in Java

If a **class** has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the **program**.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as `a(int,int)` for two parameters, and `b(int,int,int)` for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java



1. By changing number of arguments
2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first `add()` method performs addition of two numbers and second `add` method performs addition of three numbers.

In this example, we are creating **static methods** so that we don't need to create instance for calling methods.

```
class Adder{  
    static int add(int a,int b){return a+b;}  
    static int add(int a,int b,int c){return a+b+c;}  
}
```

```
class TestOverloading1{  
    public static void main(String[] args){  
        System.out.println(Adder.add(11,11));  
        System.out.println(Adder.add(11,11,11));  
    }  
}
```

Test it Now

Output:

```
22  
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in **data type**. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{  
    static int add(int a, int b){return a+b;}  
    static double add(double a, double b){return a+b;}  
}  
  
class TestOverloading2{  
    public static void main(String[] args){  
        System.out.println(Adder.add(11,11));  
        System.out.println(Adder.add(12.3,12.6));  
    }  
}
```

Test it Now

Output:

```
22  
24.9
```

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
    static int add(int a,int b){return a+b;}
    static double add(int a,int b){return a+b;}
}

class TestOverloading3{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));//ambiguity
    }
}
```

Test it Now

Output:

Compile Time Error: method add(int,int) is already defined in class Adder

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But **JVM** calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
    public static void main(String[] args){System.out.println("main with String[]");}
    public static void main(String args){System.out.println("main with String");}
    public static void main(){System.out.println("main without args");}
}
```

}

Test it Now

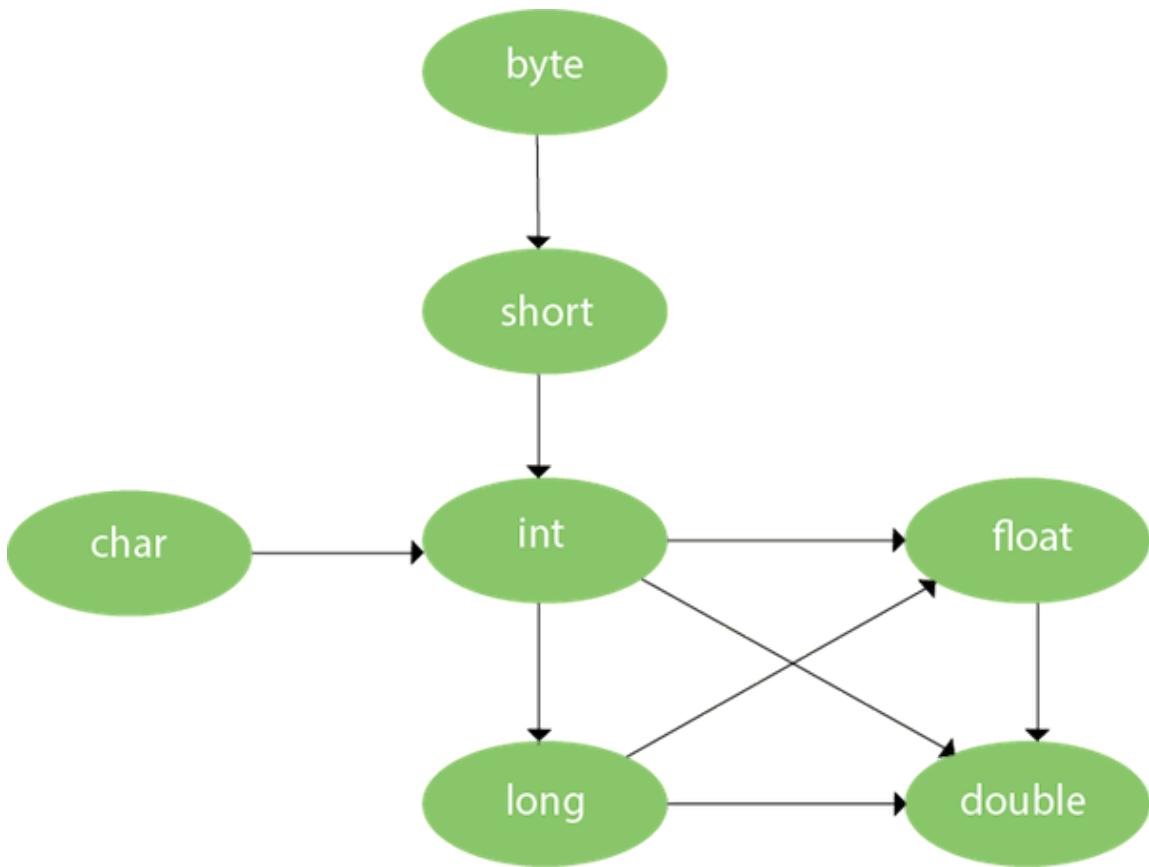
Output:

main with String[]

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:

AD



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on.

Example of Method Overloading with TypePromotion

```

class OverloadingCalculation1{
    void sum(int a,long b){System.out.println(a+b);}
    void sum(int a,int b,int c){System.out.println(a+b+c);}

    public static void main(String args[]){
        OverloadingCalculation1 obj=new OverloadingCalculation1();
        obj.sum(20,20);//now second int literal will be promoted to long
        obj.sum(20,20,20);
    }
}

```

Test it Now

Output:40
60

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```

class OverloadingCalculation2{
    void sum(int a,int b){System.out.println("int arg method invoked");}
    void sum(long a,long b){System.out.println("long arg method invoked");}

    public static void main(String args[]){
        OverloadingCalculation2 obj=new OverloadingCalculation2();
        obj.sum(20,20);//now int arg sum() method gets invoked
    }
}

```

Test it Now

Output:int arg method invoked

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class OverloadingCalculation3{  
    void sum(int a,long b){System.out.println("a method invoked");}  
    void sum(long a,int b){System.out.println("b method invoked");}  
  
    public static void main(String args[]){  
        OverloadingCalculation3 obj=new OverloadingCalculation3();  
        obj.sum(20,20); //now ambiguity  
    }  
}
```

Test it Now

Output:Compile Time Error

One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

← Prev

Next →

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Rules for Java Method Overriding



Method must have same name as in the parent class

STEP
01

Method must have same parameter as in the parent class.

There must be IS-A relationship (inheritance).

STEP
03

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.

//Creating a parent class
class Vehicle{
    void run(){System.out.println("Vehicle is running");}
}

//Creating a child class
class Bike extends Vehicle{
    public static void main(String args[]){
        //creating an instance of child class
        Bike obj = new Bike();
        //calling the method with child class instance
        obj.run();
    }
}
```

```
}
```

Test it Now

Output:

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
    //defining a method
    void run(){System.out.println("Vehicle is running");}
}

//Creating a child class
class Bike2 extends Vehicle{
    //defining the same method as in the parent class
    void run(){System.out.println("Bike is running safely");}
}

public static void main(String args[]){
    Bike2 obj = new Bike2(); //creating object
    obj.run(); //calling method
}
```

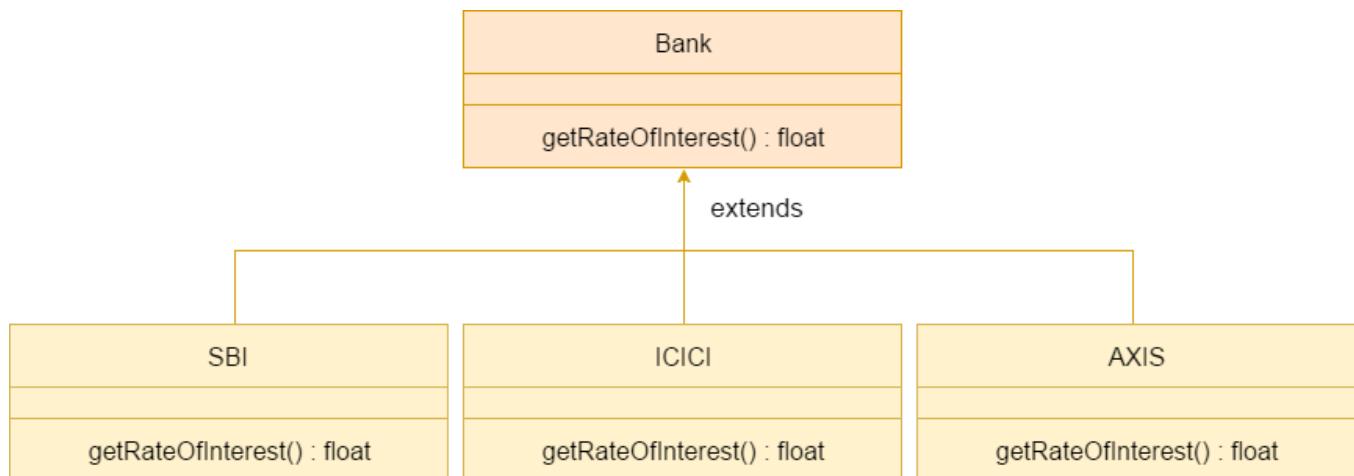
Test it Now

Output:

Bike is running safely

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

```
//Java Program to demonstrate the real scenario of Java Method Overriding
```

```
//where three classes are overriding the method of a parent class.
```

```
//Creating a parent class.
```

```
class Bank{
    int getRateOfInterest(){return 0;}
}
```

```
//Creating child classes.
```

```
class SBI extends Bank{
    int getRateOfInterest(){return 8;}
}
```

```
class ICICI extends Bank{
    int getRateOfInterest(){return 7;}
}

class AXIS extends Bank{
```

```
int getRateOfInterest(){return 9;}  
}  
  
//Test class to create objects and call the methods  
class Test2{  
    public static void main(String args[]){  
        SBI s=new SBI();  
        ICICI i=new ICICI();  
        AXIS a=new AXIS();  
        System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());  
        System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());  
        System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());  
    }  
}
```

Test it Now

Output:

```
SBI Rate of Interest: 8  
ICICI Rate of Interest: 7  
AXIS Rate of Interest: 9
```

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Difference between method Overloading and Method Overriding in java

[Click me for the difference between method overloading and overriding](#)

More topics on Method Overriding (Not For Beginners)

[Method Overriding with Access Modifier](#)

Let's see the concept of method overriding with access modifier.

[Exception Handling with Method Overriding](#)

Let's see the concept of method overriding with exception handling.

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

Simple example of Covariant Return Type

FileName: B1.java

```
class A{
A get(){return this;}
}

class B1 extends A{
@Override
B1 get(){return this;}
void message(){System.out.println("welcome to covariant return type");}

public static void main(String args[]){
new B1().get().message();
}
}
```

Test it Now

Output:

```
welcome to covariant return type
```

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

Advantages of Covariant Return Type

Following are the advantages of the covariant return type.

- 1) Covariant return type assists to stay away from the confusing type casts in the class hierarchy and makes the code more usable, readable, and maintainable.
- 2) In the method overriding, the covariant return type provides the liberty to have more to the point return types.
- 3) Covariant return type helps in preventing the run-time *ClassCastException*s on returns.

Let's take an example to understand the advantages of the covariant return type.

FileName: CovariantExample.java

```
class A1
{
    A1 foo()
    {
        return this;
    }

    void print()
    {
        System.out.println("Inside the class A1");
    }
}

// A2 is the child class of A1
class A2 extends A1
{
    @Override
    A1 foo()
    {
        return this;
    }
}
```

```
}
```

```
void print()
{
    System.out.println("Inside the class A2");
}
```

// A3 is the child class of A2

```
class A3 extends A2
```

```
{
    @Override
    A1 foo()
    {
        return this;
    }
}
```

```
@Override
void print()
{
    System.out.println("Inside the class A3");
}
```

```
public class CovariantExample
```

```
{
    // main method
    public static void main(String args[])
    {
        A1 a1 = new A1();
    }
}
```

// this is ok

```
a1.foo().print();
```

```
A2 a2 = new A2();
```

// we need to do the type casting to make it

// more clear to reader about the kind of object created

```

((A2)a2.foo()).print();

A3 a3 = new A3();

// doing the type casting
((A3)a3.foo()).print();

}

}

```

Output:

```

Inside the class A1
Inside the class A2
Inside the class A3

```

Explanation: In the above program, class A3 inherits class A2, and class A2 inherits class A1. Thus, A1 is the parent of classes A2 and A3. Hence, any object of classes A2 and A3 is also of type A1. As the return type of the method *foo()* is the same in every class, we do not know the exact type of object the method is actually returning. We can only deduce that returned object will be of type A1, which is the most generic class. We can not say for sure that returned object will be of A2 or A3. It is where we need to do the typecasting to find out the specific type of object returned from the method *foo()*. It not only makes the code verbose; it also requires precision from the programmer to ensure that typecasting is done properly; otherwise, there are fair chances of getting the *ClassCastException*. To exacerbate it, think of a situation where the hierarchical structure goes down to 10 - 15 classes or even more, and in each class, the method *foo()* has the same return type. That is enough to give a nightmare to the reader and writer of the code.

The better way to write the above is:

FileName: CovariantExample.java

```

class A1
{
    A1 foo()
    {
        return this;
    }
}

```

```
void print()
{
    System.out.println("Inside the class A1");
}
```

// A2 is the child class of A1

```
class A2 extends A1
```

```
{
    @Override
    A2 foo()
    {
        return this;
    }
}
```

```
void print()
{
    System.out.println("Inside the class A2");
}
```

// A3 is the child class of A2

```
class A3 extends A2
```

```
{
    @Override
    A3 foo()
    {
        return this;
    }
}
```

```
@Override
void print()
{
    System.out.println("Inside the class A3");
}
```

```
public class CovariantExample
{
    // main method
    public static void main(String args[])
    {
        A1 a1 = new A1();

        a1.foo().print();

        A2 a2 = new A2();

        a2.foo().print();

        A3 a3 = new A3();

        a3.foo().print();
    }
}
```

Output:

```
Inside the class A1
Inside the class A2
Inside the class A3
```

Explanation: In the above program, no typecasting is needed as the return type is specific. Hence, there is no confusion about knowing the type of object getting returned from the method `foo()`. Also, even if we write the code for the 10 - 15 classes, there would be no confusion regarding the return types of the methods. All this is possible because of the covariant return type.



How is Covariant return types implemented?

Java doesn't allow the return type-based overloading, but JVM always allows return type-based overloading. JVM uses the full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.

Output:

```
The number 1 is not the powerful number.  
The number 2 is not the powerful number.  
The number 3 is not the powerful number.  
The number 4 is the powerful number.  
The number 5 is not the powerful number.  
The number 6 is not the powerful number.  
The number 7 is not the powerful number.  
The number 8 is the powerful number.  
The number 9 is the powerful number.  
The number 10 is not the powerful number.  
The number 11 is not the powerful number.  
The number 12 is not the powerful number.  
The number 13 is not the powerful number.  
The number 14 is not the powerful number.  
The number 15 is not the powerful number.  
The number 16 is the powerful number.  
The number 17 is not the powerful number.  
The number 18 is not the powerful number.  
The number 19 is not the powerful number.  
The number 20 is the powerful number.
```

Explanation: For every number from 1 to 20, the method *isPowerfulNo()* is invoked with the help of for-loop. For every number, a vector *primeFactors* is created for storing its prime divisors. Then, we check whether square of every number present in the vector *primeFactors* divides the number or not. If all square of all the number present in the vector *primeFactors* divides the number completely, the number is a powerful number; otherwise, not.

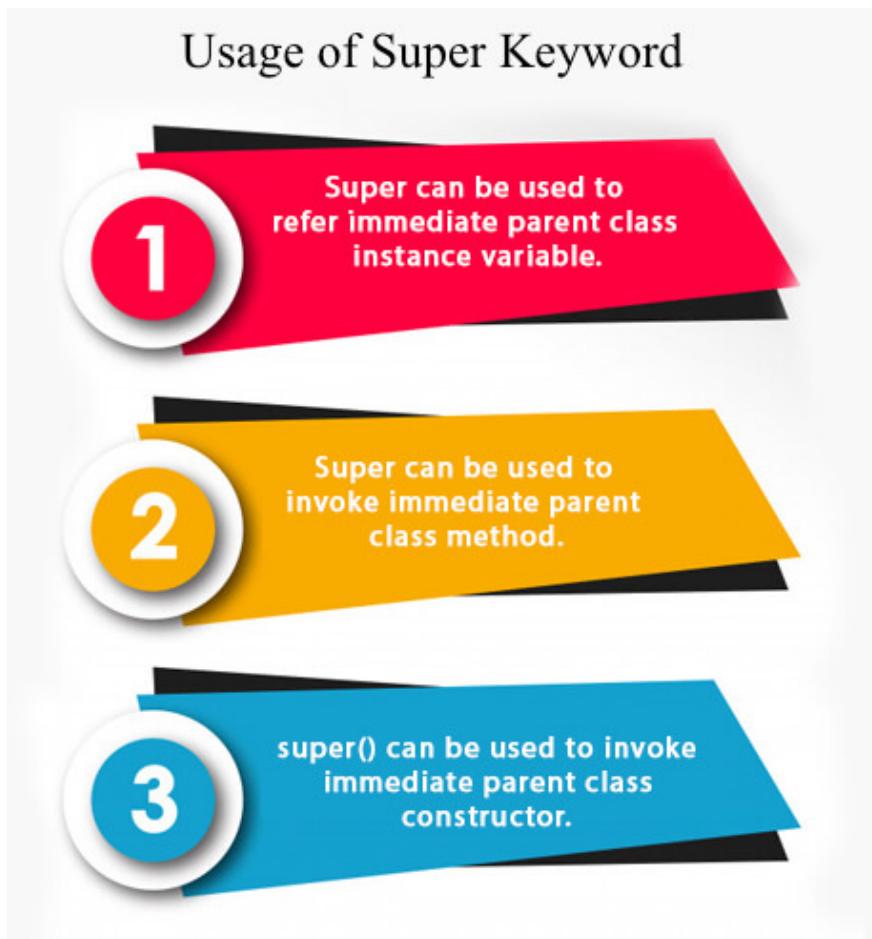
Super Keyword in Java

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.

```

class Animal{
    String color="white";
}

class Dog extends Animal{
    String color="black";
    void printColor(){
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
    }
}

class TestSuper1{
    public static void main(String args[]){
        Dog d=new Dog();
        d.printColor();
    }
}

```

Test it Now

Output:

```

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.

```

class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
    void bark(){System.out.println("barking...");}
}

```

```

void work(){
    super.eat();
    bark();
}
}

class TestSuper2{
    public static void main(String args[]){
        Dog d=new Dog();
        d.work();
    }
}

```

Test it Now

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:

```

class Animal{
    Animal(){System.out.println("animal is created");}
}

class Dog extends Animal{
    Dog(){
        super();
        System.out.println("dog is created");
    }
}

```

```

}

}

class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}

```

Test it Now

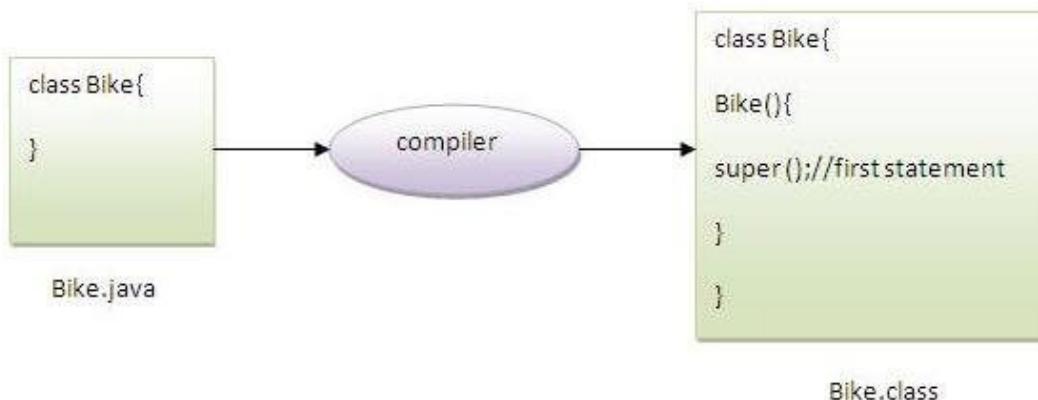
Output:

```

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.

```

class Animal{
Animal(){System.out.println("animal is created");}
}

class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}

```

```

}
}

class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}
}

```

Test it Now

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.

```

class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}

class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}

class TestSuper5{
public static void main(String[] args){
}
}

```

```
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
{}
```

Test it Now

Output:

```
1 ankit 45000
```

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

Instance initializer block

Instance Initializer block is used to initialize the instance data member. It runs each time when object of the class is created.

The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Ques) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
class Bike{  
    int speed=100;  
}
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Example of instance initializer block

Let's see the simple example of instance initializer block that performs initialization.

```
class Bike7{  
    int speed;  
  
    Bike7(){System.out.println("speed is "+speed);}  
  
    {speed=100;}  
  
    public static void main(String args[]){  
        Bike7 b1=new Bike7();  
        Bike7 b2=new Bike7();  
    }  
}
```

Test it Now

```
Output:speed is 100
      speed is 100
```

There are three places in java where you can perform operations:

1. method
2. constructor
3. block

What is invoked first, instance initializer block or constructor?

```
class Bike8{
    int speed;

    Bike8(){System.out.println("constructor is invoked");}
    {System.out.println("instance initializer block invoked");}

    public static void main(String args[]){
        Bike8 b1=new Bike8();
        Bike8 b2=new Bike8();
    }
}
```

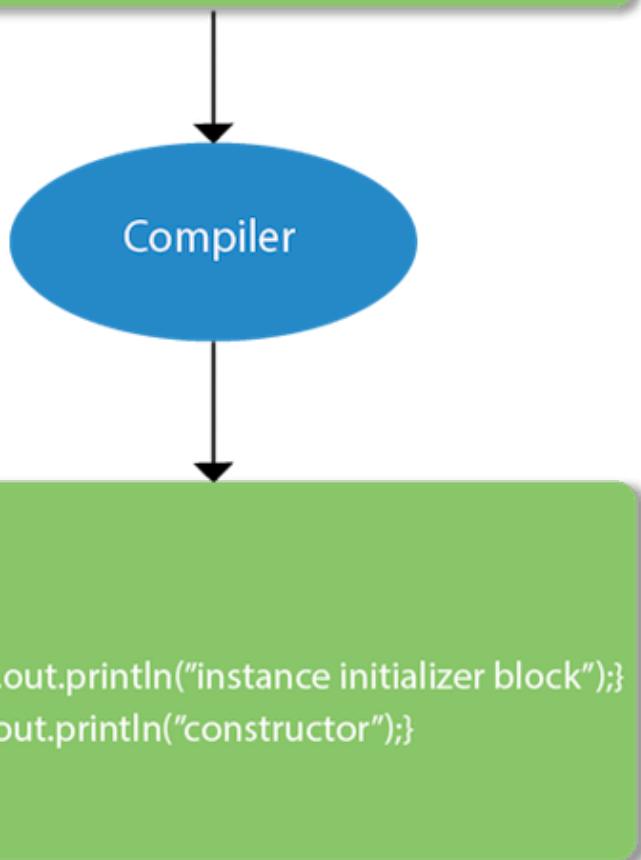
Test it Now

```
Output:instance initializer block invoked
      constructor is invoked
      instance initializer block invoked
      constructor is invoked
```

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance initializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.

```
class B{  
    B(){  
        System.out.println("constructor");  
    }  
    {System.out.println("instance initializer block");}  
}
```



Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after `super()` constructor call).
3. The instance initializer block comes in the order in which they appear.

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Program of instance initializer block that is invoked after super()

```
class A{  
    A(){  
        System.out.println("parent class constructor invoked");  
    }  
}  
  
class B2 extends A{  
    B2(){  
        super();  
        System.out.println("child class constructor invoked");  
    }  
  
    {System.out.println("instance initializer block is invoked");}  
}
```

```
public static void main(String args[]){  
    B2 b=new B2();  
}
```

Test it Now

```
Output:parent class constructor invoked  
       instance initializer block is invoked  
       child class constructor invoked
```

Another example of instance block

```
class A{  
    A(){  
        System.out.println("parent class constructor invoked");  
    }
```

```
}
```

```
class B3 extends A{  
    B3(){  
        super();  
        System.out.println("child class constructor invoked");  
    }  
  
    B3(int a){  
        super();  
        System.out.println("child class constructor invoked "+a);  
    }  
  
{System.out.println("instance initializer block is invoked");}
```

```
public static void main(String args[]){  
    B3 b1=new B3();  
    B3 b2=new B3(10);  
}
```

Test it Now

```
parent class constructor invoked  
instance initializer block is invoked  
child class constructor invoked  
parent class constructor invoked  
instance initializer block is invoked  
child class constructor invoked 10
```

← Prev

Next →

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

javatpoint.com

```
class Bike9{  
    final int speedlimit=90;//final variable  
    void run(){  
        speedlimit=400;  
    }  
    public static void main(String args[]){  
        Bike9 obj=new Bike9();  
        obj.run();  
    }  
}//end of class
```

Test it Now

Output:Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
    final void run(){System.out.println("running");}
}

class Honda extends Bike{
    void run(){System.out.println("running safely with 100kmph");}
}

public static void main(String args[]){
    Honda honda= new Honda();
    honda.run();
}
```

Test it Now

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
    void run(){System.out.println("running safely with 100kmph");}
}
```

```
public static void main(String args[]){
    Honda1 honda= new Honda1();
    honda.run();
}
```

Test it Now

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
    final void run(){System.out.println("running...");}
}

class Honda2 extends Bike{
    public static void main(String args[]){
        new Honda2().run();
    }
}
```

Test it Now

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{  
    int id;  
    String name;  
    final String PAN_CARD_NUMBER;  
    ...  
}
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{  
    final int speedlimit;//blank final variable  
  
    Bike10(){  
        speedlimit=70;  
        System.out.println(speedlimit);  
    }  
  
    public static void main(String args[]){  
        new Bike10();  
    }  
}
```

Test it Now

Output: 70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{  
    static final int data;//static blank final variable
```

```
static{ data=50;}  
public static void main(String args[]){  
    System.out.println(A.data);  
}  
}
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{  
    int cube(final int n){  
        n=n+2;//can't be changed as n is final  
        n*n*n;  
    }  
    public static void main(String args[]){  
        Bike11 b=new Bike11();  
        b.cube(5);  
    }  
}
```

Test it Now

Output: Compile Time Error

Q) Can we declare a constructor final?

No, because constructor is never inherited.

← Prev

Next →

Polymorphism in Java

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

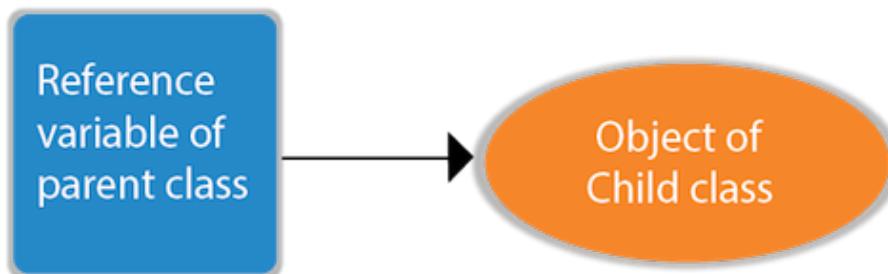
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
class A{}  
class B extends A{}
```

```
A a=new B(); //upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{}  
class A{}  
class B extends A implements I{}
```

Here, the relationship of B class would be:

```
B IS-A A  
B IS-A I  
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{  
    void run(){System.out.println("running");}  
}  
  
class Splendor extends Bike{  
    void run(){System.out.println("running safely with 60km");}  
  
    public static void main(String args[]){  
        Bike b = new Splendor(); //upcasting  
        b.run();  
    }  
}
```

}

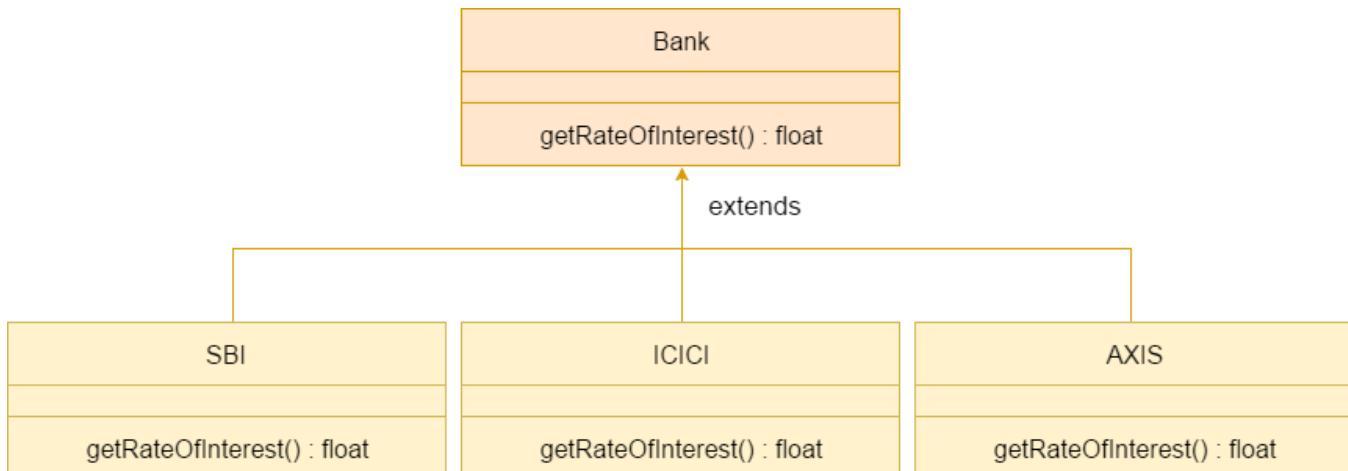
Test it Now

Output:

running safely with 60km.

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

```

class Bank{
    float getRateOfInterest(){return 0;}
}

class SBI extends Bank{
    float getRateOfInterest(){return 8.4f;}
}

class ICICI extends Bank{
    float getRateOfInterest(){return 7.3f;}
}

class AXIS extends Bank{
    float getRateOfInterest(){return 9.7f;}
}
  
```

```
}
```

```
class TestPolymorphism{
```

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

```
    Bank b;
```

```
    b=new SBI();
```

```
    System.out.println("SBI Rate of Interest: " +b.getRateOfInterest());
```

```
    b=new ICICI();
```

```
    System.out.println("ICICI Rate of Interest: " +b.getRateOfInterest());
```

```
    b=new AXIS();
```

```
    System.out.println("AXIS Rate of Interest: " +b.getRateOfInterest());
```

```
}
```

```
}
```

Test it Now

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Java Runtime Polymorphism Example: Shape

```
class Shape{
```

```
void draw(){System.out.println("drawing...");}
```

```
}
```

```
class Rectangle extends Shape{
```

```
void draw(){System.out.println("drawing rectangle...");}
```

```
}
```

```
class Circle extends Shape{
```

```
void draw(){System.out.println("drawing circle...");}
```

```
}
```

```
class Triangle extends Shape{
```

```
void draw(){System.out.println("drawing triangle...");}
```

```
}
```

```
class TestPolymorphism2{
```

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

```
Shape s;  
s=new Rectangle();  
s.draw();  
s=new Circle();  
s.draw();  
s=new Triangle();  
s.draw();  
}  
}
```

Test it Now

Output:

```
drawing rectangle...  
drawing circle...  
drawing triangle...
```

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Java Runtime Polymorphism Example: Animal

```
class Animal{  
void eat(){System.out.println("eating...");}  
}  
  
class Dog extends Animal{  
void eat(){System.out.println("eating bread...");}  
}  
  
class Cat extends Animal{  
void eat(){System.out.println("eating rat...");}  
}  
  
class Lion extends Animal{  
void eat(){System.out.println("eating meat...");}  
}  
  
class TestPolymorphism3{  
public static void main(String[] args){
```

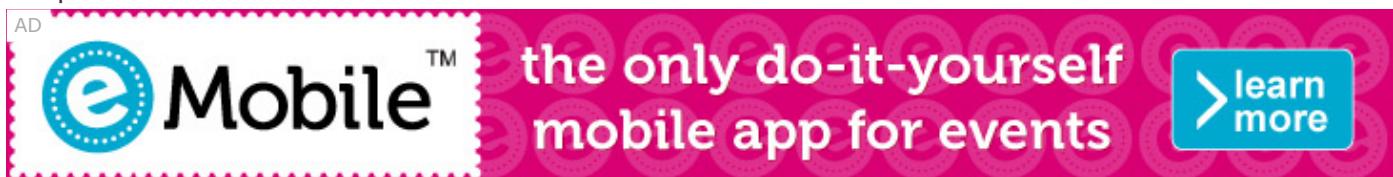
```

Animal a;
a=new Dog();
a.eat();
a=new Cat();
a.eat();
a=new Lion();
a.eat();
}}

```

[Test it Now](#)

Output:



```

eating bread...
eating rat...
eating meat...

```

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```

class Bike{
    int speedlimit=90;
}
class Honda3 extends Bike{
    int speedlimit=150;
}

```

```
public static void main(String args[]){
    Bike obj=new Honda3();
    System.out.println(obj.speedlimit);//90
}
```

Test it Now

Output:

```
90
```

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
class Animal{
    void eat(){System.out.println("eating");}
}

class Dog extends Animal{
    void eat(){System.out.println("eating fruits");}
}

class BabyDog extends Dog{
    void eat(){System.out.println("drinking milk");}
}

public static void main(String args[]){
    Animal a1,a2,a3;
    a1=new Animal();
    a2=new Dog();
    a3=new BabyDog();
    a1.eat();
    a2.eat();
    a3.eat();
}
```

Test it Now

Output:

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

```
eating  
eating fruits  
drinking Milk
```

Try for Output

```
class Animal{  
    void eat(){System.out.println("animal is eating...");}  
}  
  
class Dog extends Animal{  
    void eat(){System.out.println("dog is eating...");}  
}  
  
class BabyDog1 extends Dog{  
    public static void main(String args[]){  
        Animal a=new BabyDog1();  
        a.eat();  
    }  
}
```

Test it Now

Output:

```
Dog is eating
```

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

← Prev

Next →

Static Binding and Dynamic Binding

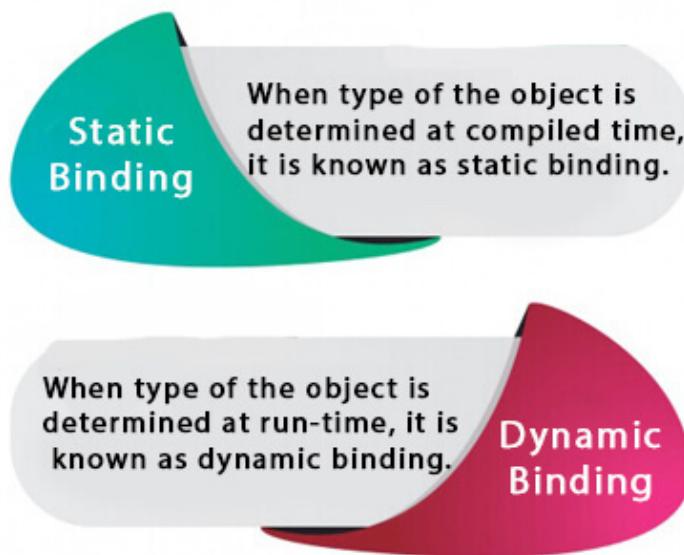
Connecting a method call to the method body is known as binding.

There are two types of binding



1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

Static vs Dynamic Binding



Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
int data=30;
```

Here data variable is a type of int.

2) References have a type

```
class Dog{
    public static void main(String args[]){
        Dog d1;//Here d1 is a type of Dog
    }
}
```

3) Objects have a type

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{}

class Dog extends Animal{
    public static void main(String args[]){
        Dog d1=new Dog();
    }
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
class Dog{
    private void eat(){System.out.println("dog is eating...");}
}

public static void main(String args[]){
    Dog d1=new Dog();
    d1.eat();
}
```

}

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{
    void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
    void eat(){System.out.println("dog is eating...");}
}

public static void main(String args[]){
    Animal a=new Dog();
    a.eat();
}
```

Test it Now

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

<<Prev

Next>>

Java instanceof

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.

```
class Simple1{
    public static void main(String args[]){
        Simple1 s=new Simple1();
        System.out.println(s instanceof Simple1);//true
    }
}
```

Test it Now

Output:true

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

Another example of java instanceof operator

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

    public static void main(String args[]){
        Dog1 d=new Dog1();
        System.out.println(d instanceof Animal);//true
    }
}
```

Test it Now

Output:true

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
class Dog2{
    public static void main(String args[]){
        Dog2 d=null;
        System.out.println(d instanceof Dog2);//false
    }
}
```

Test it Now

Output:false

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

```

class Animal { }

class Dog3 extends Animal {
    static void method(Animal a) {
        if(a instanceof Dog3){
            Dog3 d=(Dog3)a;//downcasting
            System.out.println("ok downcasting performed");
        }
    }
}

public static void main (String [] args) {
    Animal a=new Dog3();
    Dog3.method(a);
}
}

```

Test it Now

Output:ok downcasting performed

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```

class Animal { }

class Dog4 extends Animal {
    static void method(Animal a) {
        Dog4 d=(Dog4)a;//downcasting
        System.out.println("ok downcasting performed");
    }
}

public static void main (String [] args) {
    Animal a=new Dog4();
    Dog4.method(a);
}

```

```
}
```

Test it Now

```
Output:ok downcasting performed
```

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

```
Animal a=new Animal();
Dog.method(a);
//Now ClassCastException but not in case of instanceof operator
```

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

```
interface Printable{}

class A implements Printable{
    public void a(){System.out.println("a method");}
}

class B implements Printable{
    public void b(){System.out.println("b method");}
}

class Call{
    void invoke(Printable p){//upcasting
        if(p instanceof A){
            A a=(A)p;//Downcasting
            a.a();
        }
        if(p instanceof B){
            B b=(B)p;//Downcasting
            b.b();
        }
    }
}
```

```
//end of Call class
```

```
class Test4{  
    public static void main(String args[]){  
        Printable p=new B();  
        Call c=new Call();  
        c.invoke(p);  
    }  
}
```

Test it Now

Output: b method

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in **Java**. It can have abstract and non-abstract methods (method with the body).

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

Abstraction in Java

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

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

Abstraction lets you focus on what the **object** does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

Abstract class in Java

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

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have **constructors** and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Rules for Java Abstract class



- 1 An abstract class must be declared with an abstract keyword.
- 2 It can have abstract and non-abstract methods.
- 3 It cannot be instantiated.
- 4 It can have final methods.
- 5 It can have constructors and static methods also.

Example of abstract class

```
abstract class A{}
```

Abstract Method in Java

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

Example of abstract method

```
abstract void printStatus(); //no method body and abstract
```

Example of Abstract class that has an abstract method

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

```
abstract class Bike{
    abstract void run();
}

class Honda4 extends Bike{
    void run(){System.out.println("running safely");}
    public static void main(String args[]){
        Bike obj = new Honda4();
        obj.run();
    }
}
```

Test it Now

running safely

Understanding the real scenario of Abstract class

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

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

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

File: TestAbstraction1.java

```
abstract class Shape{
    abstract void draw();
}

//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}

class Circle1 extends Shape{
```

```

void draw(){System.out.println("drawing circle");}
}

//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() met
s.draw();
}
}

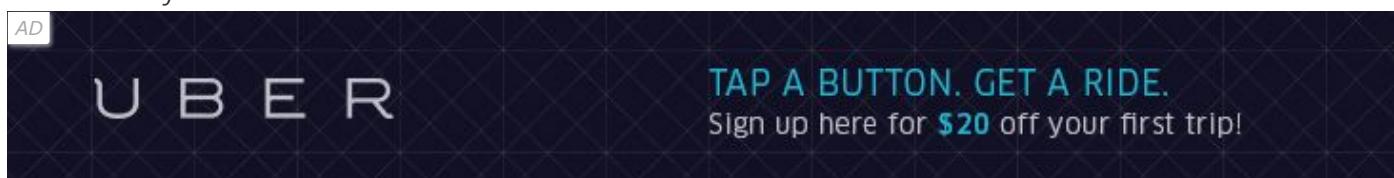
```

Test it Now

drawing circle

Another example of Abstract class in java

File: TestBank.java



```

abstract class Bank{
abstract int getRateOfInterest();
}

class SBI extends Bank{
int getRateOfInterest(){return 7;}
}

class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
}
}

```

```
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Test it Now

```
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

```
//Example of an abstract class that has abstract and non-abstract methods
abstract class Bike{
    Bike(){System.out.println("bike is created");}
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}

//Creating a Child class which inherits Abstract class
class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}

//Creating a Test class which calls abstract and non-abstract methods
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

Test it Now

```
bike is created
running safely..
```

gear changed

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

```
class Bike12{  
    abstract void run();  
}
```

Test it Now

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.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the **interface**. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
interface A{  
    void a();  
    void b();  
    void c();  
    void d();  
}  
  
abstract class B implements A{  
    public void c(){System.out.println("I am c");}  
}
```

```
class M extends B{  
    public void a(){System.out.println("I am a");}  
    public void b(){System.out.println("I am b");}  
    public void d(){System.out.println("I am d");}  
}
```

```
class Test5{  
    public static void main(String args[]){  
        A a=new M();  
        a.a();  
        a.b();  
        a.c();  
        a.d();  
    }  
}
```

Test it Now

Output:I am a
I am b
I am c
I am d

← Prev

Next →

AD



Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple **inheritance in Java**.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.



How to declare an interface?

An interface is declared by using the `interface` keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

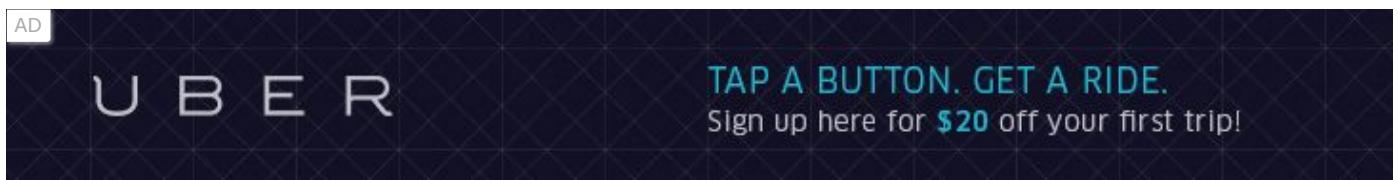
```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.

}
```

Java 8 Interface Improvement

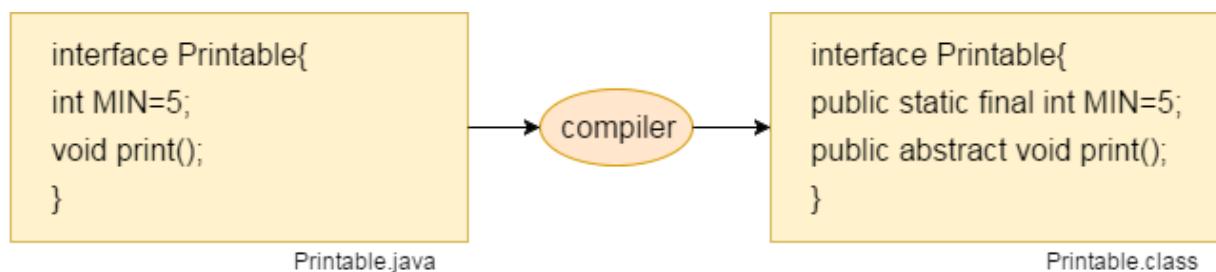
Since [Java 8](#), interface can have default and static methods which is discussed later.



Internal addition by the compiler

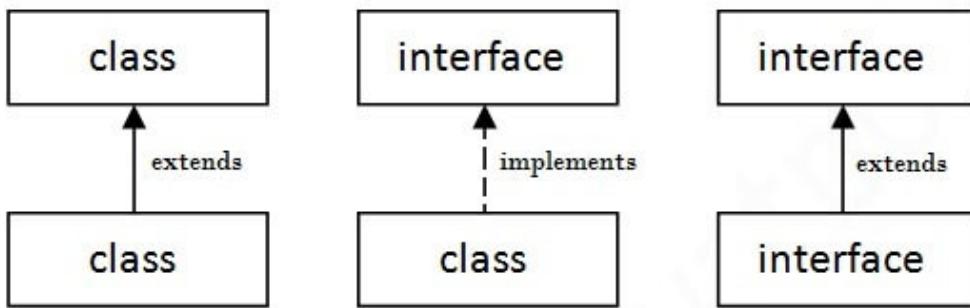
The Java compiler adds `public` and `abstract` keywords before the interface method. Moreover, it adds `public`, `static` and `final` keywords before data members.

In other words, Interface fields are `public`, `static` and `final` by default, and the methods are `public` and `abstract`.



The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the `Printable` interface has only one method, and its implementation is provided in the `A6` class.

```
interface printable{
    void print();
}

class A6 implements printable{
    public void print(){System.out.println("Hello");}
}

public static void main(String args[]){
    A6 obj = new A6();
    obj.print();
}
```

Test it Now

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
//Interface declaration: by first user
interface Drawable{
    void draw();
}

//Implementation: by second user
class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}

class Circle implements Drawable{
    public void draw(){System.out.println("drawing circle");}
}

//Using interface: by third user
class TestInterface1{
    public static void main(String args[]){
        Drawable d=new Circle(); //In real scenario, object is provided by method e.g. getDrawable()
        d.draw();
    }
}
```

Test it Now

Output:



```
drawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```

interface Bank{
    float rateOfInterest();
}

class SBI implements Bank{
    public float rateOfInterest(){return 9.15f;}
}

class PNB implements Bank{
    public float rateOfInterest(){return 9.7f;}
}

class TestInterface2{
    public static void main(String[] args){
        Bank b=new SBI();
        System.out.println("ROI: " + b.rateOfInterest());
    }
}

```

Test it Now

Output:

ROI: 9.15

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

**Multiple Inheritance in Java**

```

interface Printable{
void print();
}

interface Showable{
void show();
}

class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}

```

Test it Now

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** 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:



```

interface Printable{
void print();
}

interface Showable{
void print();
}

```

```
}
```

```
class TestInterface3 implements Printable, Showable{
    public void print(){System.out.println("Hello");}
    public static void main(String args[]){
        TestInterface3 obj = new TestInterface3();
        obj.print();
    }
}
```

Test it Now

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestInterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
    void print();
}

interface Showable extends Printable{
    void show();
}

class TestInterface4 implements Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}
}

public static void main(String args[]){
    TestInterface4 obj = new TestInterface4();
    obj.print();
    obj.show();
}
```

```
}
```

Test it Now

Output:

```
Hello  
Welcome
```

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```
interface Drawable{  
    void draw();  
    default void msg(){System.out.println("default method");}  
}  
  
class Rectangle implements Drawable{  
    public void draw(){System.out.println("drawing rectangle");}  
}  
  
class TestInterfaceDefault{  
    public static void main(String args[]){  
        Drawable d=new Rectangle();  
        d.draw();  
        d.msg();  
    }  
}
```

Test it Now

Output:

```
drawing rectangle  
default method
```

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
interface Drawable{
    void draw();
    static int cube(int x){return x*x*x;}
}

class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
    public static void main(String args[]){
        Drawable d=new Rectangle();
        d.draw();
        System.out.println(Drawable(cube(3)));
    }
}
```

Test it Now

Output:

```
drawing rectangle
27
```

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, **Serializable**, **Cloneable**, **Remote**, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{}
```

Nested Interface in Java

Note: An interface can have another interface which is known as a nested interface. We will learn it in detail in the [nested classes](#) chapter. For example:

```
interface printable{
    void print();
}

interface MessagePrintable{
    void msg();
}
```

More about Nested Interface

« Prev

Next »

AD

 [YouTube](#) For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- o Send your Feedback to feedback@javatpoint.com

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods  
interface A{  
    void a();  
    void b();  
    void c();  
    void d();  
}
```

```
//Creating abstract class that provides the implementation of one method of A interface  
abstract class B implements A{  
    public void c(){System.out.println("I am C");}  
}
```

```
//Creating subclass of abstract class, now we need to provide the implementation of rest of the me  
class M extends B{  
    public void a(){System.out.println("I am a");}  
    public void b(){System.out.println("I am b");}  
    public void d(){System.out.println("I am d");}  
}
```

```
//Creating a test class that calls the methods of A interface  
class Test5{  
    public static void main(String args[]){  
        A a=new M();  
        a.a();  
        a.b();  
        a.c();  
        a.d();  
    }  
}
```

Test it Now

Output:

```
I am a  
I am b  
I am c  
I am d
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

[Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

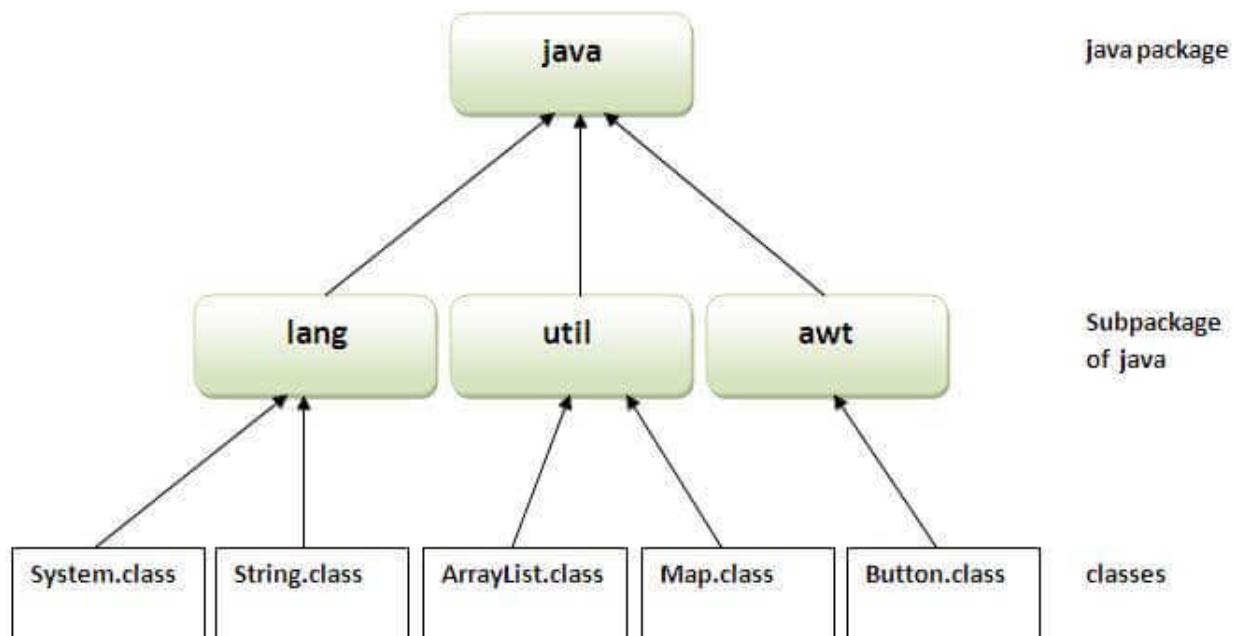
Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
```

```
public class Simple{  
    public static void main(String args[]){  
        System.out.println("Welcome to package");  
    }  
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For **example**

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

AD

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

```
Output:Hello
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

AD

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

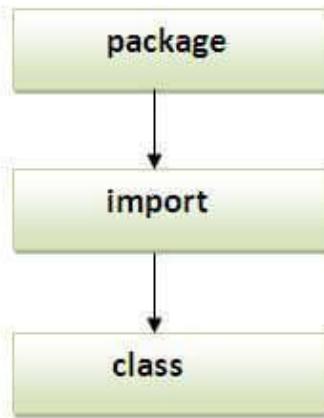
```
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A(); //using fully qualified name
        obj.msg();
    }
}
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has defined a package named `java` that contains many classes like `System`, `String`, `Reader`, `Writer`, `Socket` etc. These classes represent a particular group e.g. `Reader` and `Writer` classes are for Input/Output operation, `Socket` and `ServerSocket` classes are for networking etc and so on. So, Sun has subcategorized the `java` package into subpackages such as `lang`, `net`, `io` etc. and put the Input/Output related classes in `io` package, `Server` and `ServerSocket` classes in `net` packages and so on.

The standard of defining package is `domain.company.package` e.g. `com.javatpoint.bean` or `org.sssit.dao`.

Example of Subpackage

```

package com.javatpoint.core;
class Simple{
    public static void main(String args[]){
        System.out.println("Hello subpackage");
    }
}
  
```

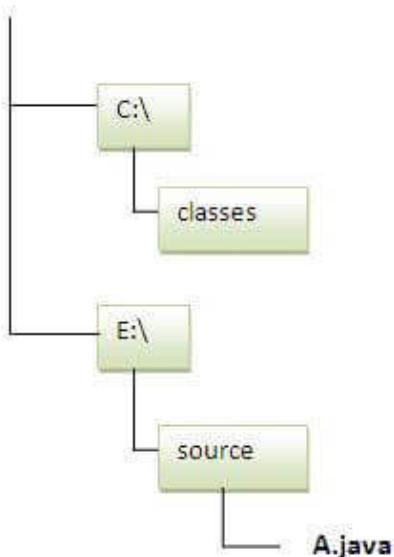
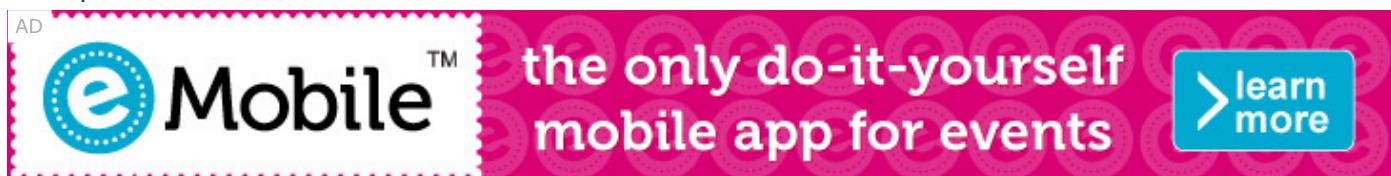
To Compile: `javac -d . Simple.java`

To Run: `java com.javatpoint.core.Simple`

Output:Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:



```
//save as Simple.java
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

To Compile:

```
e:\sources> javac -d c:\classes Simple.java
```

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

```
e:\sources> set classpath=c:\classes;.
```

```
e:\sources> java mypack.Simple
```

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
 - By setting the classpath in the command prompt
 - By -classpath switch
- Permanent
 - By setting the classpath in the environment variables
 - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

//save as C.java otherwise Compile Time Error

```
class A{}  
class B{}  
public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
//save as A.java
```

```
package javatpoint;  
public class A{}
```

```
//save as B.java
```

```
package javatpoint;  
public class B{}
```

What is static import feature of Java5?

Click [Static Import](#) feature of Java5.

What about package class?

Click for [Package class](#)

← Prev

Next →

Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

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.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

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

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

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

```
class A{
    private int data=40;
    private void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();
        System.out.println(obj.data);//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

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:

```
class A{
    private A(){}//private constructor
    void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();//Compile Time Error
    }
}
```

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.

```
//save by A.java
package pack;
class A{
    void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

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.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B extends A{
public static void main(String args[]){
B obj = new B();
obj.msg();
}
}
```

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

```
//save by A.java

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

Java Access Modifiers with Method Overriding

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

```
class A{
    protected void msg(){System.out.println("Hello java");}
}
```

```
public class Simple extends A{  
    void msg(){System.out.println("Hello java");}//C.T.Error  
    public static void main(String args[]){  
        Simple obj=new Simple();  
        obj.msg();  
    }  
}
```

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



← Prev

Next →

AD

[Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

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

```
//A Java class which is a fully encapsulated class.  
//It has a private data member and getter and setter methods.  
package com.javatpoint;  
public class Student{  
    //private data member  
    private String name;  
    //getter method for name
```

```
public String getName(){
    return name;
}

//setter method for name
public void setName(String name){
    this.name=name
}
```

File: Test.java

```
//A Java class to test the encapsulated class.

package com.javatpoint;
class Test{
    public static void main(String[] args){
        //creating instance of the encapsulated class
        Student s=new Student();
        //setting value in the name member
        s.setName("vijay");
        //getting value of the name member
        System.out.println(s.getName());
    }
}
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
```

Output:

```
vijay
```

Read-Only class

```
//A Java class which has only getter methods.

public class Student{
    //private data member
```

```
private String college="AKG";
//getter method for college
public String getCollege(){
    return college;
}
}
```

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

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

Write-Only class

```
//A Java class which has only setter methods.
public class Student{
    //private data member
    private String college;
    //getter method for college
    public void setCollege(String college){
        this.college=college;
    }
}
```

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

```
System.out.println(s.getCollege());//Compile Time Error, because there is no such method
System.out.println(s.college);//Compile Time Error, because the college data member is private.
//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

```
//A Account class which is a fully encapsulated class.  
//It has a private data member and getter and setter methods.  
class Account {  
    //private data members  
    private long acc_no;  
    private String name,email;  
    private float amount;  
    //public getter and setter methods  
    public long getAcc_no() {  
        return acc_no;  
    }  
    public void setAcc_no(long acc_no) {  
        this.acc_no = acc_no;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getEmail() {  
        return email;  
    }  
    public void setEmail(String email) {  
        this.email = email;  
    }  
    public float getAmount() {  
        return amount;  
    }  
    public void setAmount(float amount) {  
        this.amount = amount;  
    }  
}
```

File: TestAccount.java

```
//A Java class to test the encapsulated class Account.  
public class TestEncapsulation {  
    public static void main(String[] args) {  
        //creating instance of Account class  
        Account acc=new Account();  
        //setting values through setter methods  
        acc.setAcc_no(7560504000L);  
        acc.setName("Sonoo Jaiswal");  
        acc.setEmail("sonoojaiswal@javatpoint.com");  
        acc.setAmount(500000f);  
        //getting values through getter methods  
        System.out.println(acc.getAcc_no() + " " + acc.getName() + " " + acc.getEmail() + " " + acc.getAmount());  
    }  
}
```

Test it Now

Output:

```
7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0
```

← Prev

Next →

Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

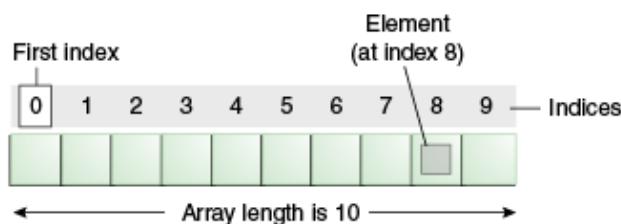
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

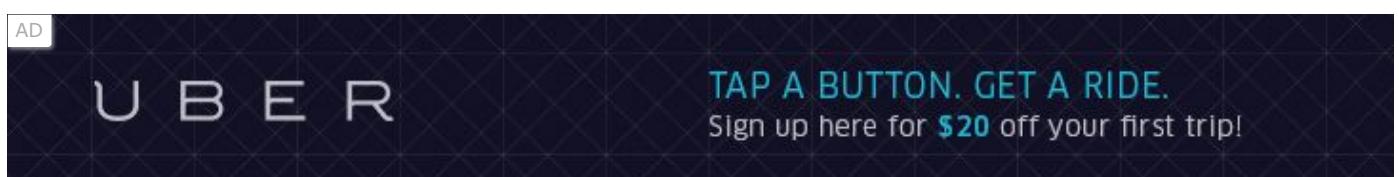


Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.



Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[] arr; (or)  
dataType []arr; (or)  
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
//Java Program to illustrate how to declare, instantiate, initialize  
//and traverse the Java array.  
class Testarray{  
    public static void main(String args[]){  
        int a[]={new int[5]};  
        //declaration and instantiation  
        a[0]=10;  
        //initialization  
        a[1]=20;  
        a[2]=70;  
        a[3]=40;  
        a[4]=50;  
        //traversing array  
        for(int i=0;i<a.length;i++)  
            //length is the property of array  
            System.out.println(a[i]);  
    }  
}
```

```
}
```

Test it Now

Output:

```
10  
20  
70  
40  
50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation  
//and initialization of Java array in a single line  
  
class Testarray1{  
    public static void main(String args[]){  
        int a[]={33,3,4,5};//declaration, instantiation and initialization  
        //printing array  
        for(int i=0;i<a.length;i++)//length is the property of array  
            System.out.println(a[i]);  
    }  
}
```

Test it Now

Output:

```
33  
3
```

4

5

For-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){  
    //body of the loop  
}
```

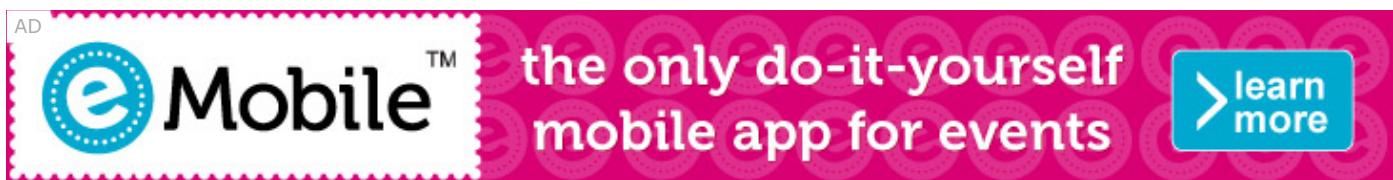
Let us see the example of print the elements of Java array using the for-each loop.



```
//Java Program to print the array elements using for-each loop  
  
class Testarray1{  
    public static void main(String args[]){  
        int arr[]={33,3,4,5};  
        //printing array using for-each loop  
        for(int i:arr)  
            System.out.println(i);  
    }  
}
```

Output:

```
33  
3  
4  
5
```



Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

```
//Java Program to demonstrate the way of passing an array  
//to method.  
class Testarray2{  
    //creating a method which receives an array as a parameter  
    static void min(int arr[]){  
        int min=arr[0];  
        for(int i=1;i<arr.length;i++)  
            if(min>arr[i])  
                min=arr[i];  
  
        System.out.println(min);  
    }  
  
public static void main(String args[]){  
    int a[]={33,3,4,5};//declaring and initializing an array  
    min(a);//passing array to method  
}
```

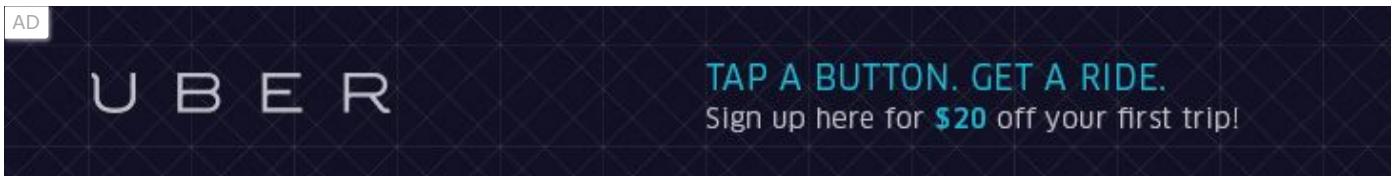
Test it Now

Output:

```
3
```

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.



```
//Java Program to demonstrate the way of passing an anonymous array  
//to method.  
public class TestAnonymousArray{  
    //creating a method which receives an array as a parameter  
    static void printArray(int arr[]){  
        for(int i=0;i<arr.length;i++)  
            System.out.println(arr[i]);  
    }  
  
public static void main(String args[]){  
    printArray(new int[]{10,22,44,66});//passing anonymous array to method  
}
```

Test it Now

Output:

```
10  
22  
44  
66
```

Returning Array from the Method

We can also return an array from the method in Java.

```
//Java Program to return an array from the method  
class TestReturnArray{  
    //creating method which returns an array  
    static int[] get(){  
        return new int[]{10,30,50,90,60};  
    }  
}
```

```
public static void main(String args[]){
    //calling method which returns an array
    int arr[] = get();
    //printing the values of an array
    for(int i=0; i<arr.length; i++)
        System.out.println(arr[i]);
}
```

Test it Now

Output:

```
10
30
50
90
60
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an `ArrayIndexOutOfBoundsException` if length of the array is negative, equal to the array size or greater than the array size while traversing the array.

```
//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.
public class TestArrayException{
    public static void main(String args[]){
        int arr[] = {50, 60, 70, 80};
        for(int i=0; i<=arr.length; i++)
            System.out.println(arr[i]);
    }
}
```

Test it Now

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at TestArrayException.main(TestArrayException.java:5)
50
60
70
80
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
    public static void main(String args[]){
        //declaring and initializing 2D array
        int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
        //printing 2D array
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();
        }
    }
}
```

Test it Now

Output:

```
1 2 3
2 4 5
4 4 5
```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

```
//Java Program to illustrate the jagged array
class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
```

```

arr[2] = new int[2];
//initializing a jagged array
int count = 0;
for (int i=0; i<arr.length; i++)
    for(int j=0; j<arr[i].length; j++)
        arr[i][j] = count++;

//printing the data of a jagged array
for (int i=0; i<arr.length; i++){
    for (int j=0; j<arr[i].length; j++){
        System.out.print(arr[i][j]+ " ");
    }
    System.out.println();//new line
}
}
}

```

Test it Now

Output:

```

0 1 2
3 4 5 6
7 8

```

What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by `getClass().getName()` method on the object.

```

//Java Program to get the class name of array in Java
class Testarray4{
public static void main(String args[]){
//declaration and initialization of array
int arr[]={4,4,5};
//getting the class name of Java array
Class c=arr.getClass();
String name=c.getName();
}
}

```

```
//printing the class name of Java array  
System.out.println(name);  
  
}}
```

Test it Now

Output:

```
I
```

Copying a Java Array

We can copy an array to another by the `arraycopy()` method of `System` class.

Syntax of `arraycopy` method

```
public static void arraycopy(  
Object src, int srcPos, Object dest, int destPos, int length  
)
```

Example of Copying an Array in Java

```
//Java Program to copy a source array into a destination array in Java  
class TestArrayCopyDemo {  
    public static void main(String[] args) {  
        //declaring a source array  
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
                           'i', 'n', 'a', 't', 'e', 'd' };  
        //declaring a destination array  
        char[] copyTo = new char[7];  
        //copying array using System.arraycopy() method  
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
        //printing the destination array  
        System.out.println(String.valueOf(copyTo));  
    }  
}
```

```
}
```

Test it Now

Output:

```
caffein
```

Cloning an Array in Java

Since, Java array implements the `Cloneable` interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

```
//Java Program to clone the array
class Testarray1{
    public static void main(String args[]){
        int arr[]={33,3,4,5};
        System.out.println("Printing original array:");
        for(int i:arr)
            System.out.println(i);

        System.out.println("Printing clone of the array:");
        int carr[]=arr.clone();
        for(int i:carr)
            System.out.println(i);

        System.out.println("Are both equal?");
        System.out.println(arr==carr);
    }
}
```

Output:

```
Printing original array:
33
```

```
3  
4  
5  
Printing clone of the array:  
33  
3  
4  
5  
Are both equal?  
false
```

Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

```
//Java Program to demonstrate the addition of two matrices in Java  
class Testarray5{  
    public static void main(String args[]){  
        //creating two matrices  
        int a[][]={{1,3,4},{3,4,5}};  
        int b[][]={{1,3,4},{3,4,5}};  
  
        //creating another matrix to store the sum of two matrices  
        int c[][]=new int[2][3];  
  
        //adding and printing addition of 2 matrices  
        for(int i=0;i<2;i++){  
            for(int j=0;j<3;j++){  
                c[i][j]=a[i][j]+b[i][j];  
                System.out.print(c[i][j] + " ");  
            }  
            System.out.println();//new line  
        }  
    }  
}
```

Test it Now

Output:

```
2 6 8
6 8 10
```

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.

$$\begin{array}{l}
 \text{Matrix 1} \left\{ \begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{array} \right\} \quad \text{Matrix 2} \left\{ \begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{array} \right\} \\
 \\
 \text{Matrix 1} * \text{Matrix 2} \left\{ \begin{array}{ccc} 1*1+1*2+1*3 & 1*1+1*2+1*3 & 1*1+1*2+1*3 \\ 2*1+2*2+2*3 & 2*1+2*2+2*3 & 2*1+2*2+2*3 \\ 3*1+3*2+3*3 & 3*1+3*2+3*3 & 3*1+3*2+3*3 \end{array} \right\} \\
 \\
 \text{Matrix 1} * \text{Matrix 2} \left\{ \begin{array}{ccc} 6 & 6 & 6 \\ 12 & 12 & 12 \\ 18 & 18 & 18 \end{array} \right\}
 \end{array}$$

JavaTpoint

Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
    public static void main(String args[]){
        //creating two matrices
        int a[][]={{1,1,1},{2,2,2},{3,3,3}};
        int b[][]={{1,1,1},{2,2,2},{3,3,3}};

        //creating another matrix to store the multiplication of two matrices
        int c[][]=new int[3][3]; //3 rows and 3 columns

        //multiplying and printing multiplication of 2 matrices
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                c[i][j]=0;
                for(int k=0;k<3;k++)
                    c[i][j] += a[i][k]*b[k][j];
            }
        }

        //printing the result
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++)
                System.out.print(c[i][j] + " ");
            System.out.println();
        }
    }
}
```

```
c[i][j]=0;  
for(int k=0;k<3;k++)  
{  
    c[i][j]+=a[i][k]*b[k][j];  
}//end of k loop  
System.out.print(c[i][j]+" "); //printing matrix element  
}//end of j loop  
System.out.println();//new line  
}  
}  
}}
```

Test it Now

Output:

```
6 6 6  
12 12 12  
18 18 18
```

Related Topics

- 1) Java Program to copy all elements of one array into another array
- 2) Java Program to find the frequency of each element in the array
- 3) Java Program to left rotate the elements of an array
- 4) Java Program to print the duplicate elements of an array
- 5) Java Program to print the elements of an array
- 6) Java Program to print the elements of an array in reverse order
- 7) Java Program to print the elements of an array present on even position
- 8) Java Program to print the elements of an array present on odd position
- 9) Java Program to print the largest element in an array
- 10) Java Program to print the smallest element in an array
- 11) Java Program to print the number of elements present in an array
- 12) Java Program to print the sum of all the items of the array
- 13) Java Program to right rotate the elements of an array
- 14) Java Program to sort the elements of an array in ascending order
- 15) Java Program to sort the elements of an array in descending order
- 16) Find 3rd Largest Number in an Array
- 17) Find 2nd Largest Number in an Array
- 18) Find Largest Number in an Array
- 19) Find 2nd Smallest Number in an Array
- 20) Find Smallest Number in an Array
- 21) Remove Duplicate Element in an Array
- 22) Add Two Matrices
- 23) Multiply Two Matrices
- 24) Print Odd and Even Number from an Array
- 25) Transpose matrix
- 26) Java Program to subtract the two matrices
- 27) Java Program to determine whether a given matrix is an identity matrix
- 28) Java Program to determine whether a given matrix is a sparse matrix
- 29) Java Program to determine whether two matrices are equal

- 30) Java Program to display the lower triangular matrix
- 31) Java Program to display the upper triangular matrix
- 32) Java Program to find the frequency of odd & even numbers in the given matrix
- 33) Java Program to find the product of two matrices
- 34) Java Program to find the sum of each row and each column of a matrix
- 35) Java Program to find the transpose of a given matrix

[← Prev](#)[Next →](#)

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 Splunk tutorial

Splunk

 SPSS tutorial

SPSS



Swagger tutorial

Swagger

 T-SQL tutorial

Transact-SQL

Object class in Java

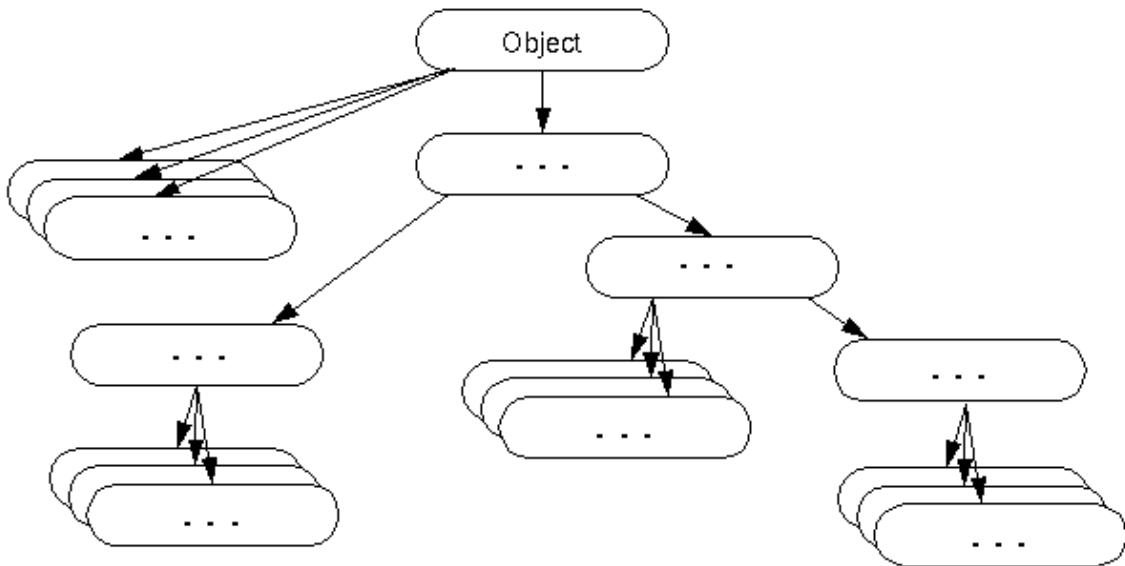
The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

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

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

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

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



Methods of Object class

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

Method	Description
<code>public final Class getClass()</code>	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
<code>public int hashCode()</code>	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.

We will have the detailed learning of these methods in next chapters.

← Prev

Next →

Object Cloning in Java

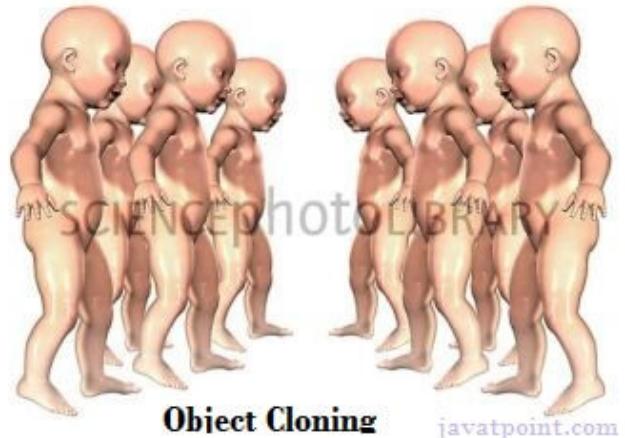
The **object cloning** is a way to create exact copy of an object. The `clone()` method of `Object` class is used to clone an object.

The **`java.lang.Cloneable` interface** must be implemented by the class whose object clone we want to create. If we don't implement `Cloneable` interface, `clone()` method generates **`CloneNotSupportedException`**.

The **`clone()` method** is defined in the `Object` class.

Syntax of the `clone()` method is as follows:

```
protected Object clone() throws CloneNotSupportedException
```



Why use `clone()` method ?

The **`clone()` method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the `new` keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Advantage of Object cloning

Although `Object.clone()` has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using `clone()` method:

- You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long `clone()` method.
- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement `Cloneable` in it, provide the definition of the `clone()` method and the task will be done.
- `Clone()` is the fastest way to copy array.

Disadvantage of Object cloning

Following is a list of some disadvantages of `clone()` method:

- To use the `Object.clone()` method, we have to change a lot of syntaxes to our code, like implementing a `Cloneable` interface, defining the `clone()` method and handling `CloneNotSupportedException`, and finally, calling `Object.clone()` etc.

- We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
- If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.



Example of clone() method (Object cloning)

Let's see the simple example of object cloning

```
class Student18 implements Cloneable{
    int rollno;
    String name;

    Student18(int rollno, String name){
        this.rollno = rollno;
        this.name = name;
    }

    public Object clone() throws CloneNotSupportedException{
        return super.clone();
    }

    public static void main(String args[]){
        try{
            Student18 s1 = new Student18(101, "amit");

            Student18 s2 = (Student18)s1.clone();
        }
    }
}
```

```
System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){

}

}
```

Test it Now

```
Output:101 amit
      101 amit
```

download the example of object cloning

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

← Prev

Next →

AD

 For Videos Join Our Youtube Channel: [Join Now](#)

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.

Example 1

```
public class JavaMathExample1
{
    public static void main(String[] args)
    {
        double x = 28;
        double y = 4;

        // return the maximum of two numbers
        System.out.println("Maximum number of x and y is: " + Math.max(x, y));

        // return the square root of y
        System.out.println("Square root of y is: " + Math.sqrt(y));

        //returns 28 power of 4 i.e. 28*28*28*28
        System.out.println("Power of x and y is: " + Math.pow(x, y));

        // return the logarithm of given value
        System.out.println("Logarithm of x is: " + Math.log(x));
        System.out.println("Logarithm of y is: " + Math.log(y));
    }
}
```

```
// return the logarithm of given value when base is 10
System.out.println("log10 of x is: " + Math.log10(x));
System.out.println("log10 of y is: " + Math.log10(y));

// return the log of x + 1
System.out.println("log1p of x is: " + Math.log1p(x));

// return a power of 2
System.out.println("exp of a is: " + Math.exp(x));

// return (a power of 2)-1
System.out.println("expm1 of a is: " + Math.expm1(x));
}

}
```

Test it Now**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

```
public class JavaMathExample2
{
    public static void main(String[] args)
    {
        double a = 30;
```

```
// converting values to radian
double b = Math.toRadians(a);

// return the trigonometric sine of a
System.out.println("Sine value of a is: " +Math.sin(a));

// return the trigonometric cosine value of a
System.out.println("Cosine value of a is: " +Math.cos(a));

// return the trigonometric tangent value of a
System.out.println("Tangent value of a is: " +Math.tan(a));

// return the trigonometric arc sine of a
System.out.println("Sine value of a is: " +Math.asin(a));

// return the trigonometric arc cosine value of a
System.out.println("Cosine value of a is: " +Math.acos(a));

// return the trigonometric arc tangent value of a
System.out.println("Tangent value of a is: " +Math.atan(a));

// return the hyperbolic sine of a
System.out.println("Sine value of a is: " +Math.sinh(a));

// return the hyperbolic cosine value of a
System.out.println("Cosine value of a is: " +Math.cosh(a));

// return the hyperbolic tangent value of a
System.out.println("Tangent value of a is: " +Math.tanh(a));
}
```

Test it Now

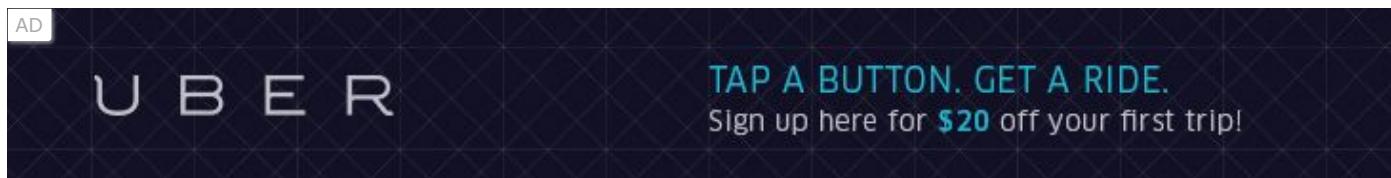
Output:

```
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 Math Methods

The **java.lang.Math** class contains various methods for performing basic numeric operations such as the logarithm, cube root, and trigonometric functions etc. The various java math methods are as follows:



Basic Math methods

Method	Description
Math.abs()	It will return the Absolute value of the given value.
Math.max()	It returns the Largest of two values.
Math.min()	It is used to return the Smallest of two values.
Math.round()	It is used to round of the decimal numbers to the nearest value.
Math.sqrt()	It is used to return the square root of a number.
Math.cbrt()	It is used to return the cube root of a number.
Math.pow()	It returns the value of first argument raised to the power to second argument.
Math.signum()	It is used to find the sign of a given value.
Math.ceil()	It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer.

<code>Math.copySign()</code>	It is used to find the Absolute value of first argument along with sign specified in second argument.
<code>Math.nextAfter()</code>	It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.
<code>Math.nextUp()</code>	It returns the floating-point value adjacent to d in the direction of positive infinity.
<code>Math.nextDown()</code>	It returns the floating-point value adjacent to d in the direction of negative infinity.
<code>Math.floor()</code>	It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value.
<code>Math.floorDiv()</code>	It is used to find the largest integer value that is less than or equal to the algebraic quotient.
<code>Math.random()</code>	It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
<code>Math.rint()</code>	It returns the double value that is closest to the given argument and equal to mathematical integer.
<code>Math.hypot()</code>	It returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
<code>Math.ulp()</code>	It returns the size of an ulp of the argument.
<code>Math.getExponent()</code>	It is used to return the unbiased exponent used in the representation of a value.
<code>Math.IEEEremainder()</code>	It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value.
<code>Math.addExact()</code>	It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long.
<code>Math.subtractExact()</code>	It returns the difference of the arguments, throwing an exception if the result overflows an int.
<code>Math.multiplyExact()</code>	It is used to return the product of the arguments, throwing an exception if the result overflows an int or long.

<code>Math.incrementExact()</code>	It returns the argument incremented by one, throwing an exception if the result overflows an int.
<code>Math.decrementExact()</code>	It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long.
<code>Math.negateExact()</code>	It is used to return the negation of the argument, throwing an exception if the result overflows an int or long.
<code>Math.toIntExact()</code>	It returns the value of the long argument, throwing an exception if the value overflows an int.

Logarithmic Math Methods

Method	Description
<code>Math.log()</code>	It returns the natural logarithm of a double value.
<code>Math.log10()</code>	It is used to return the base 10 logarithm of a double value.
<code>Math.log1p()</code>	It returns the natural logarithm of the sum of the argument and 1.
<code>Math.exp()</code>	It returns E raised to the power of a double value, where E is Euler's number and it is approximately equal to 2.71828.
<code>Math.expm1()</code>	It is used to calculate the power of E and subtract one from it.

Trigonometric Math Methods

Method	Description
<code>Math.sin()</code>	It is used to return the trigonometric Sine value of a Given double value.
<code>Math.cos()</code>	It is used to return the trigonometric Cosine value of a Given double value.
<code>Math.tan()</code>	It is used to return the trigonometric Tangent value of a Given double value.
<code>Math.asin()</code>	It is used to return the trigonometric Arc Sine value of a Given double value
<code>Math.acos()</code>	It is used to return the trigonometric Arc Cosine value of a Given double value.
<code>Math.atan()</code>	It is used to return the trigonometric Arc Tangent value of a Given double value.

Hyperbolic Math Methods

Method	Description
Math.sinh()	It is used to return the trigonometric Hyperbolic Cosine value of a Given double value.
Math.cosh()	It is used to return the trigonometric Hyperbolic Sine value of a Given double value.
Math.tanh()	It is used to return the trigonometric Hyperbolic Tangent value of a Given double value.

Angular Math Methods

Method	Description
Math.toDegrees	It is used to convert the specified Radians angle to equivalent angle measured in Degrees.
Math.toRadians	It is used to convert the specified Degrees angle to equivalent angle measured in Radians.

← Prev

Next →

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Wrapper classes in Java

The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive.*

Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer

long	Long
float	Float
double	Double

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the `valueOf()` method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
    public static void main(String args[]){
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);//converting int into Integer explicitly
        Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
        System.out.println(a+" "+i+" "+j);
    }
}
```

Output:

```
20 20 20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the `intValue()` method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int

public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

3 3 3



Java Wrapper classes Example

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa

public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;
```

```
//Autoboxing: Converting primitives into objects
```

```
Byte byteobj=b;  
Short shortobj=s;  
Integer intobj=i;  
Long longobj=l;  
Float floatobj=f;  
Double doubleobj=d;  
Character charobj=c;  
Boolean boolobj=b2;
```

```
//Printing objects
```

```
System.out.println("---Printing object values---");  
System.out.println("Byte object: "+byteobj);  
System.out.println("Short object: "+shortobj);  
System.out.println("Integer object: "+intobj);  
System.out.println("Long object: "+longobj);  
System.out.println("Float object: "+floatobj);  
System.out.println("Double object: "+doubleobj);  
System.out.println("Character object: "+charobj);  
System.out.println("Boolean object: "+boolobj);
```

```
//Unboxing: Converting Objects to Primitives
```

```
byte bytevalue=byteobj;  
short shortvalue=shortobj;  
int intvalue=intobj;  
long longvalue=longobj;  
float floatvalue=floatobj;  
double doublevalue=doubleobj;  
char charvalue=charobj;  
boolean boolvalue=boolobj;
```

```
//Printing primitives
```

```
System.out.println("---Printing primitive values---");  
System.out.println("byte value: "+bytevalue);  
System.out.println("short value: "+shortvalue);  
System.out.println("int value: "+intvalue);  
System.out.println("long value: "+longvalue);  
System.out.println("float value: "+floatvalue);
```

```
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
{}
```

Output:

```
--Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
--Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

Custom Wrapper class in Java

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

```
//Creating the custom wrapper class
class Javatpoint{
    private int i;
    Javatpoint(){}
    Javatpoint(int i){
        this.i=i;
```

```
}

public int getValue(){
return i;
}

public void setValue(int i){
this.i=i;
}

@Override

public String toString() {
return Integer.toString(i);
}

}

//Testing the custom wrapper class

public class TestJavatpoint{
public static void main(String[] args){
Javatpoint j=new Javatpoint(10);
System.out.println(j);
}}
```

Output:

```
10
```

← Prev

Next →

AD

Call by Value and Call by Reference in Java

There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{  
    int data=50;  
  
    void change(int data){  
        data=data+100;//changes will be in the local variable only  
    }  
  
    public static void main(String args[]){  
        Operation op=new Operation();  
  
        System.out.println("before change "+op.data);  
        op.change(500);  
        System.out.println("after change "+op.data);  
    }  
}
```

[download this example](#)

```
Output:before change 50  
          after change 50
```

Another Example of call by value in java

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

```
class Operation2{
    int data=50;

    void change(Operation2 op){
        op.data=op.data+100;//changes will be in the instance variable
    }

    public static void main(String args[]){
        Operation2 op=new Operation2();

        System.out.println("before change "+op.data);
        op.change(op);//passing object
        System.out.println("after change "+op.data);

    }
}
```

[download this example](#)

Output: before change 50
after change 150

← Prev

Next →

AD

 For Videos Join Our Youtube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 Splunk tutorial Splunk	 SPSS tutorial SPSS	 Swagger tutorial Swagger	 T-SQL tutorial Transact-SQL
 Tumblr tutorial Tumblr	 React tutorial ReactJS	 Regex tutorial Regex	 Reinforcement learning tutorial Reinforcement Learning
 R Programming tutorial R Programming	 RxJS tutorial RxJS	 React Native tutorial React Native	 Python Design Patterns Python Design Patterns
 Python Pillow tutorial Python Pillow	 Python Turtle tutorial Python Turtle	 Keras tutorial Keras	

Preparation

 Aptitude Aptitude	 Logical Reasoning	 Verbal Ability Verbal Ability	 Interview Questions
--	---	--	---

Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

Syntax:

```
returntype methodname(){  
    //code to be executed  
    methodname(); //calling same method  
}
```

Java Recursion Example 1: Infinite times

```
public class RecursionExample1 {  
    static void p(){  
        System.out.println("hello");  
        p();  
    }  
  
    public static void main(String[] args) {  
        p();  
    }  
}
```

Output:

```
hello  
hello  
...  
java.lang.StackOverflowError
```

Java Recursion Example 2: Finite times

```
public class RecursionExample2 {  
    static int count=0;  
    static void p(){  
        count++;  
        if(count<=5){  
            System.out.println("hello "+count);  
            p();  
        }  
    }  
    public static void main(String[] args) {  
        p();  
    }  
}
```

Output:

```
hello 1  
hello 2  
hello 3  
hello 4  
hello 5
```

Java Recursion Example 3: Factorial Number

```
public class RecursionExample3 {  
    static int factorial(int n){  
        if (n == 1)  
            return 1;  
        else  
            return(n * factorial(n-1));  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Factorial of 5 is: "+factorial(5));  
    }  
}
```

```
}
```

```
}
```

Output:

```
Factorial of 5 is: 120
```

Working of above program:

```
factorial(5)
factorial(4)
    factorial(3)
        factorial(2)
            factorial(1)
                return 1
            return 2*1 = 2
        return 3*2 = 6
    return 4*6 = 24
return 5*24 = 120
```

AD

Java Recursion Example 4: Fibonacci Series

```
public class RecursionExample4 {
    static int n1=0,n2=1,n3=0;
    static void printFibo(int count){
        if(count>0){
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;
            System.out.print(" "+n3);
            printFibo(count-1);
        }
    }
}
```

```
public static void main(String[] args) {  
    int count=15;  
    System.out.print(n1+" "+n2);//printing 0 and 1  
    printFibo(count-2);//n-2 because 2 numbers are already printed  
}  
}
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Java Strictfp Keyword

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

```
strictfp class A{}//strictfp applied on class
```

```
strictfp interface M{}//strictfp applied on interface
```

```
class A{  
    strictfp void m(){}//strictfp applied on method  
}
```

Illegal code for strictfp keyword

The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

```
class B{  
    strictfp abstract void m();//Illegal combination of modifiers  
}
```

```
class B{  
    strictfp int data=10;//modifier strictfp not allowed here  
}
```

```
class B{  
    strictfp B(){}//modifier strictfp not allowed here
```

}

[← Prev](#)[Next →](#)

AD



For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



Splunk



SPSS



Swagger tutorial

Swagger



Transact-SQL



Tumblr



ReactJS



Regex



Reinforcement learning

Creating API Document | javadoc tool

We can create document api in java by the help of **javadoc** tool. In the java file, we must use the documentation comment `/**... */` to post information for the class, method, constructor, fields etc.

Let's see the simple class that contains documentation comment.

```
package com.abc;  
/** This class is a user-defined class that contains one methods cube.*/  
public class M{  
  
    /** The cube method prints cube of the given number */  
    public static void cube(int n){System.out.println(n*n*n);}  
}
```

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

```
javadoc M.java
```

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

← Prev

Next →

Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{
    public static void main(String args[]){
        System.out.println("Your first argument is: " +args[0]);
    }
}
```

compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
class A{
    public static void main(String args[]){
        for(int i=0;i<args.length;i++)
            System.out.println(args[i]);
    }
}
```

```
}
```

```
}
```

compile by > javac A.java

run by > java A sonoo jaiswal 1 3 abc

Output: sonoo

jaiswal

1

3

abc

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

No.	Object	Class
1)	Object is an instance of a class.	Class is a blueprint or template from which objects are created.
2)	Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a group of similar objects .
3)	Object is a physical entity.	Class is a logical entity.
4)	Object is created through new keyword mainly e.g. Student s1=new Student();	Class is declared using class keyword e.g. class Student{}
5)	Object is created many times as per requirement.	Class is declared once .
6)	Object allocates memory when it is created .	Class doesn't allocated memory when it is created .
7)	There are many ways to create object in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only one way to define class in java using class keyword.

Let's see some real life example of class and object in java to understand the difference well:

Class: Human **Object:** Man, Woman

Class: Fruit **Object:** Apple, Banana, Mango, Guava wtc.

Class: Mobile phone **Object:** iPhone, Samsung, Moto

Class: Food **Object:** Pizza, Burger, Samosa

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

Java Method Overloading example

```
class OverloadingExample{
    static int add(int a,int b){return a+b;}
    static int add(int a,int b,int c){return a+b+c;}
}
```

Java Method Overriding example

```
class Animal{
void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

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.

In this tutorial, we will learn about Java exceptions, its types, and the difference between checked and unchecked exceptions.

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.

Exception Handling in Java - Javatpoint



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:

```
statement 1;  
statement 2;  
statement 3;
```

```
statement 4;  
statement 5;//exception occurs  
statement 6;  
statement 7;  
statement 8;  
statement 9;  
statement 10;
```

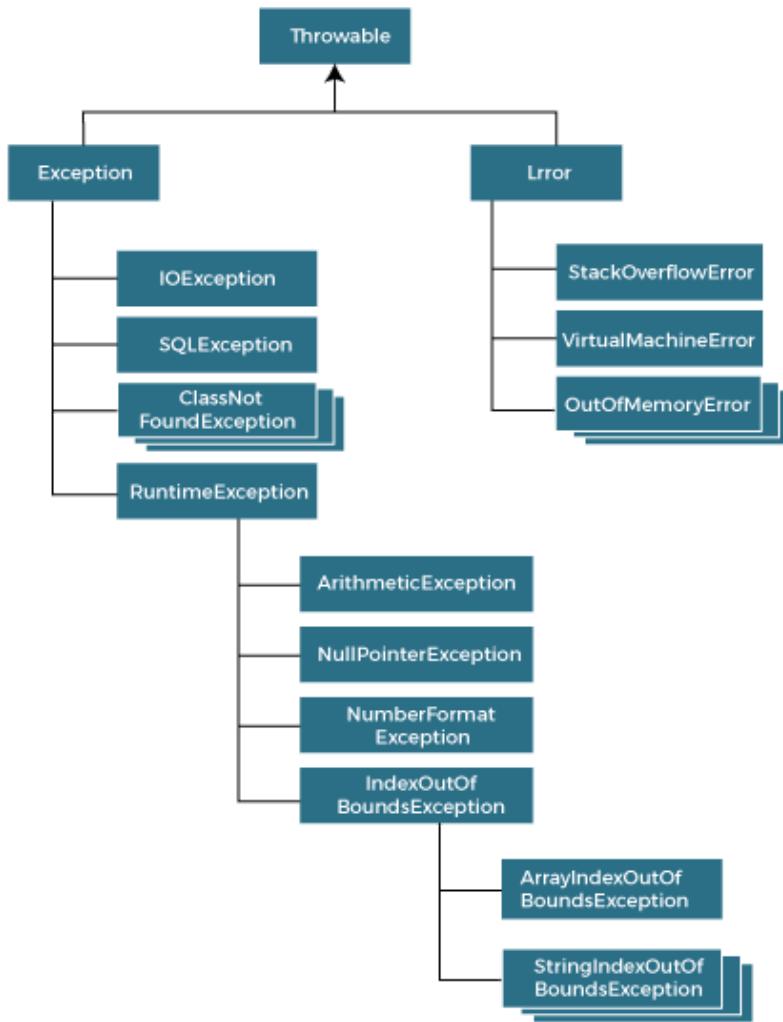
Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling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

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



AD

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, `ArithmaticException`, `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

```
public class JavaExceptionExample{
    public static void main(String args[]){
        try{
            //code that may raise exception
            int data=100/0;
        }catch(ArithmaticException e){System.out.println(e);}
        //rest code of the program
        System.out.println("rest of the code...");
    }
}
```

Test it Now

Output:

```
Exception in thread main java.lang.ArithmetricException:/ by zero  
rest of the code...
```

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

AD

Common Scenarios of Java Exceptions

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

1) A scenario where ArithmetricException occurs

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

```
int a=50/0;//ArithmetricException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;  
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 variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s="abc";  
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to its size, the ArrayIndexOutOfBoundsException occurs. There may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[] = new int[5];  
a[10] = 50; //ArrayIndexOutOfBoundsException
```

Java Exceptions Index

1. Java Try-Catch Block
2. Java Multiple Catch Block
3. Java Nested Try
4. Java Finally Block
5. Java Throw Keyword
6. Java Exception Propagation
7. Java Throws Keyword
8. Java Throw vs Throws
9. Java Final vs Finally vs Finalize
10. Java Exception Handling with Method Overriding
11. Java Custom Exceptions

← Prev

Next →

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

```
try{
    //code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

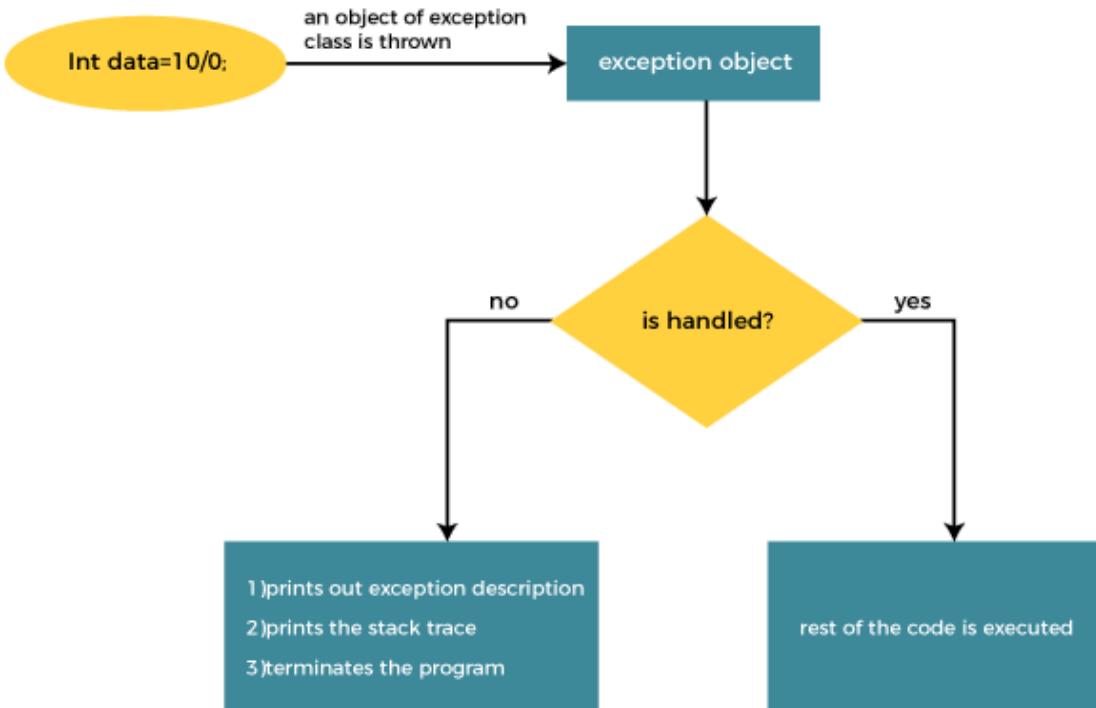
```
try{
    //code that may throw an exception
}finally{}
```

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.

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

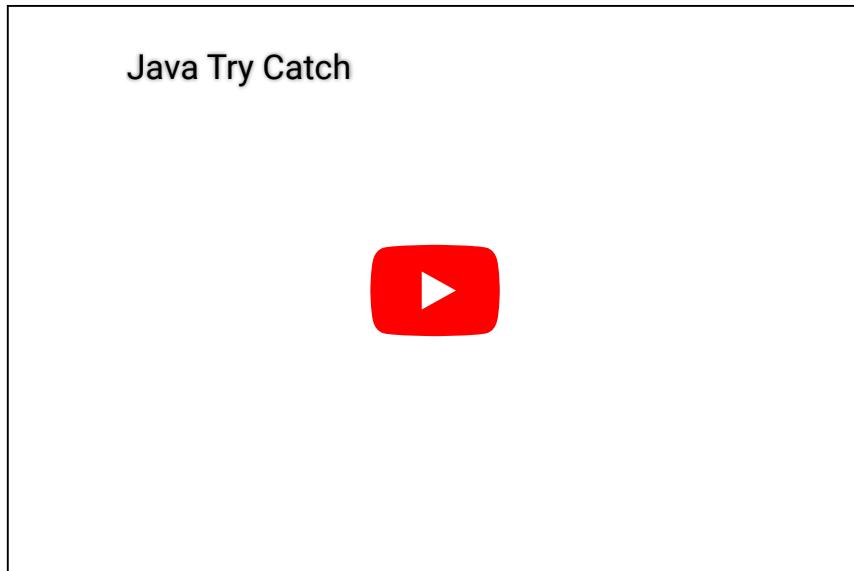
Internal Working of Java try-catch block



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:

- o Prints out exception description.
- o Prints the stack trace (Hierarchy of methods where the exception occurred).
- o 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

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

Test it Now

Output:

```
Exception in thread "main" java.lang.ArithmetiException: / 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

```
public class TryCatchExample2 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        //handling the exception  
        catch(ArithmetricException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
  
}
```

Test it Now

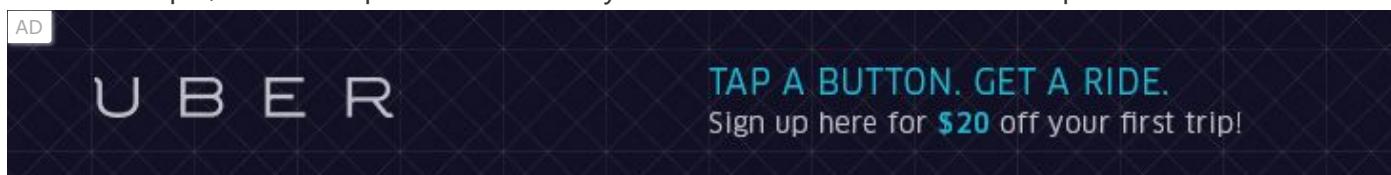
Output:

```
java.lang.ArithmetricException: / 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

```
public class TryCatchExample3 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
                // if exception occurs, the remaining statement will not execute  
            System.out.println("rest of the code");  
        }  
        // handling the exception  
        catch(ArithmetricException e)  
        {  
            System.out.println(e);  
        }  
  
    }  
  
}
```

Test it Now

Output:

```
java.lang.ArithmetricException: / 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

An advertisement for Gaiam TV, featuring the text "VIDEO STREAMING FOR THE AWAKENED MIND" and a "GET STARTED NOW" button. The background shows a scenic landscape with mountains and trees.

```
public class TryCatchExample4 {
```

```
public static void main(String[] args) {  
    try  
    {  
        int data=50/0; //may throw exception  
    }  
    // handling the exception by using Exception class  
    catch(Exception e)  
    {  
        System.out.println(e);  
    }  
    System.out.println("rest of the code");  
}  
  
}
```

Test it Now

Output:

```
java.lang.ArithmaticException: / by zero  
rest of the code
```

Example 5

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

TryCatchExample5.java

```
public class TryCatchExample5 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        // handling the exception  
        catch(Exception e)
```

```
{  
    // displaying the custom message  
    System.out.println("Can't divided by zero");  
}  
}  
  
}
```

Test it Now

Output:

```
Can't divided by zero
```

Example 6

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

TryCatchExample6.java

```
public class TryCatchExample6 {  
  
    public static void main(String[] args) {  
        int i=50;  
        int j=0;  
        int data;  
        try  
        {  
            data=i/j; //may throw exception  
        }  
        // handling the exception  
        catch(Exception e)  
        {  
            // resolving the exception in catch block  
            System.out.println(i/(j+2));  
        }  
    }  
}
```

```
}
```

Test it Now

Output:

```
25
```

Example 7

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

TryCatchExample7.java

```
public class TryCatchExample7 {  
  
    public static void main(String[] args) {  
  
        try  
        {  
            int data1=50/0; //may throw exception  
  
        }  
        // handling the exception  
        catch(Exception e)  
        {  
            // generating the exception in catch block  
            int data2=50/0; //may throw exception  
  
        }  
        System.out.println("rest of the code");  
    }  
}
```

Test it Now

Output:

```
Exception in thread "main" java.lang.ArithmetricException: / 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

```
public class TryCatchExample8 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
  
        }  
        // try to handle the ArithmetricException using ArrayIndexOutOfBoundsException  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
  
}
```

Test it Now

Output:

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

Example 9

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

TryCatchExample9.java

```
public class TryCatchExample9 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int arr[] = {1,3,5,7};  
            System.out.println(arr[10]); //may throw exception  
        }  
        // handling the array exception  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
  
}
```

Test it Now

Output:

```
java.lang.ArrayIndexOutOfBoundsException: 10  
rest of the code
```

Example 10

Let's see an example to handle checked exception.

TryCatchExample10.java

```
import java.io.FileNotFoundException;  
import java.io.PrintWriter;
```

```
public class TryCatchExample10 {  
  
    public static void main(String[] args) {  
  
        PrintWriter pw;  
        try {  
            pw = new PrintWriter("jtp.txt"); //may throw exception  
            pw.println("saved");  
        }  
        // providing the checked exception handler  
        catch (FileNotFoundException e) {  
  
            System.out.println(e);  
        }  
        System.out.println("File saved successfully");  
    }  
}
```

Test it Now

Output:

```
File saved successfully
```

← Prev

Next →

Java Catch Multiple Exceptions

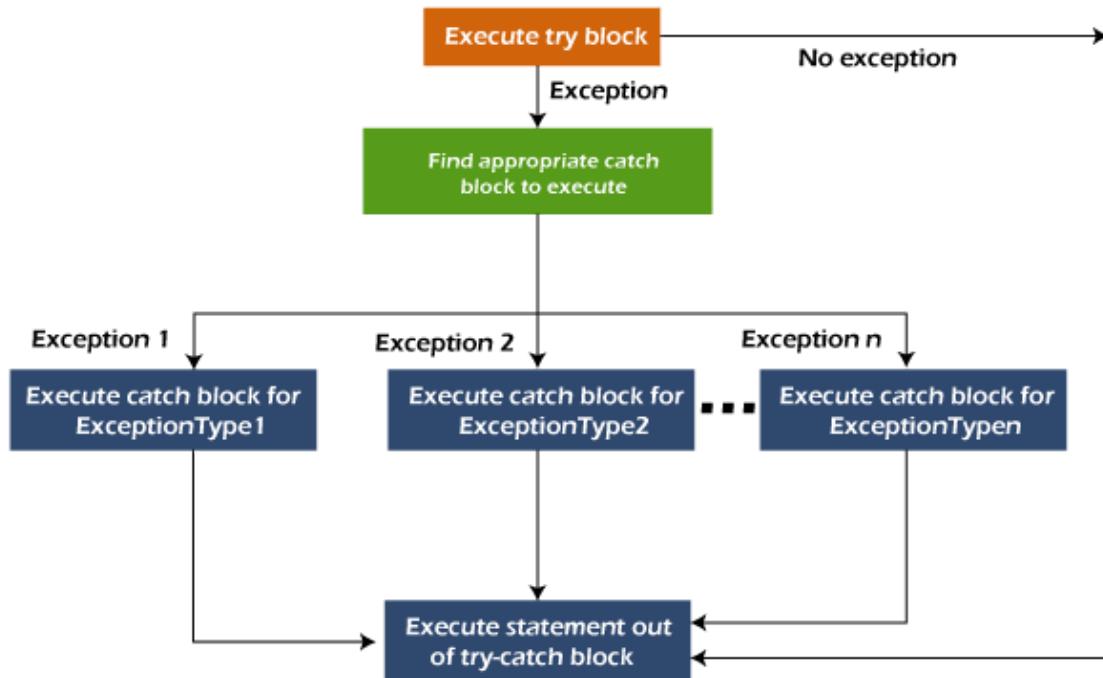
Java Multi-catch block

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 `ArithmaticException` 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

```
public class MultipleCatchBlock1 {
```

```
public static void main(String[] args) {  
  
    try{  
        int a[]={new int[5];  
        a[5]=30/0;  
    }  
    catch(ArithmeticException e)  
    {  
        System.out.println("Arithmetic Exception occurs");  
    }  
    catch(ArrayIndexOutOfBoundsException e)  
    {  
        System.out.println("ArrayIndexOutOfBoundsException occurs");  
    }  
    catch(Exception e)  
    {  
        System.out.println("Parent Exception occurs");  
    }  
    System.out.println("rest of the code");  
}  
}
```

Test it Now

Output:

```
Arithmetic Exception occurs  
rest of the code
```

Example 2

MultipleCatchBlock2.java

```
public class MultipleCatchBlock2 {  
  
    public static void main(String[] args) {  
  
        try{  
    
```

```
int a[]={new int[5];  
  
        System.out.println(a[10]);  
    }  
  
    catch(ArithmeticException e)  
    {  
        System.out.println("Arithmetic Exception occurs");  
    }  
  
    catch(ArrayIndexOutOfBoundsException e)  
    {  
        System.out.println("ArrayIndexOutOfBoundsException occurs");  
    }  
  
    catch(Exception e)  
    {  
        System.out.println("Parent Exception occurs");  
    }  
  
    System.out.println("rest of the code");  
}  
}
```

Test it Now

Output:

```
ArrayIndexOutOfBoundsException 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

```
public class MultipleCatchBlock3 {  
  
    public static void main(String[] args) {  
  
        try{  
            int a[]={new int[5];
```

```
a[5]=30/0;  
System.out.println(a[10]);  
}  
catch(ArithmaticException e)  
{  
    System.out.println("Arithmatic Exception occurs");  
}  
catch(ArrayIndexOutOfBoundsException e)  
{  
    System.out.println("ArrayIndexOutOfBoundsException occurs");  
}  
catch(Exception e)  
{  
    System.out.println("Parent Exception occurs");  
}  
System.out.println("rest of the code");  
}  
}
```

Test it Now

Output:

```
Arithmatic 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 be invoked.

MultipleCatchBlock4.java

```
public class MultipleCatchBlock4 {  
  
    public static void main(String[] args) {  
  
        try{
```

```

String s=null;
System.out.println(s.length());
}
catch(ArithmetricException e)
{
    System.out.println("Arithmetric Exception occurs");
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("ArrayIndexOutOfBoundsException occurs");
}
catch(Exception e)
{
    System.out.println("Parent Exception occurs");
}
System.out.println("rest of the code");
}
}

```

Test it Now**Output:**

```

Parent Exception occurs
rest of the code

```

Example 5

Let's see an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

MultipleCatchBlock5.java

```

class MultipleCatchBlock5{
public static void main(String args[]){
try{
int a[] = new int[5];
a[5] = 30/0;
}
}

```

```
}

catch(Exception e){System.out.println("common task completed");}

catch(ArithmaticException e){System.out.println("task1 is completed");}

catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

System.out.println("rest of the code...");

}

}
```

Test it Now

Output:

Compile-time error

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....  
//main try block  
try  
{  
    statement 1;  
    statement 2;  
    //try catch block within another try block  
    try  
    {  
        statement 3;  
        statement 4;  
        //try catch block within nested try block  
        try  
        {  
            statement 5;  
            statement 6;  
        }  
        catch(Exception e2)  
        {  
            //exception message  
        }  
    }  
}
```

```
catch(Exception e1)
{
//exception message
}
}

//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

Java Nested try Example

Example 1

Let's see an example where we place a try block within another try block for two different exceptions.

NestedTryBlock.java

```
public class NestedTryBlock{
public static void main(String args[]){
//outer try block
try{
//inner try block 1
try{
System.out.println("going to divide by 0");
int b =39/0;
}
//catch block of inner try block 1
catch(ArithmaticException e)
{
System.out.println(e);
}
}

//inner try block 2
```

```

try{
    int a[]=new int[5];

    //assigning the value out of array bounds
    a[5]=4;
}

//catch block of inner try block 2
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(e);
}

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

//catch block of outer try block
catch(Exception e)
{
    System.out.println("handled the exception (outer catch)");
}

System.out.println("normal flow..");
}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java
C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmetricException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..

```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

Example 2

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

NestedTryBlock.java

```
public class NestedTryBlock2 {  
  
    public static void main(String args[])  
    {  
        // outer (main) try block  
        try {  
  
            //inner try block 1  
            try {  
  
                // inner try block 2  
                try {  
                    int arr[] = { 1, 2, 3, 4 };  
  
                    //printing the array element out of its bounds  
                    System.out.println(arr[10]);  
                }  
  
                // to handles ArithmeticException  
                catch (ArithmaticException e) {  
                    System.out.println("Arithmatic exception");  
                    System.out.println(" inner try block 2");  
                }  
            }  
  
            // to handle ArithmaticException
```

```
catch (ArithmaticException e) {  
    System.out.println("Arithmatic exception");  
    System.out.println("inner try block 1");  
}  
  
}  
  
// to handle ArrayIndexOutOfBoundsException  
catch (ArrayIndexOutOfBoundsException e4) {  
    System.out.print(e4);  
    System.out.println(" outer (main) try block");  
}  
  
catch (Exception e5) {  
    System.out.print("Exception");  
    System.out.println(" handled in main try-block");  
}  
}  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock2.java  
C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock2  
java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 outer  
(main) try block
```

← Prev

Next →

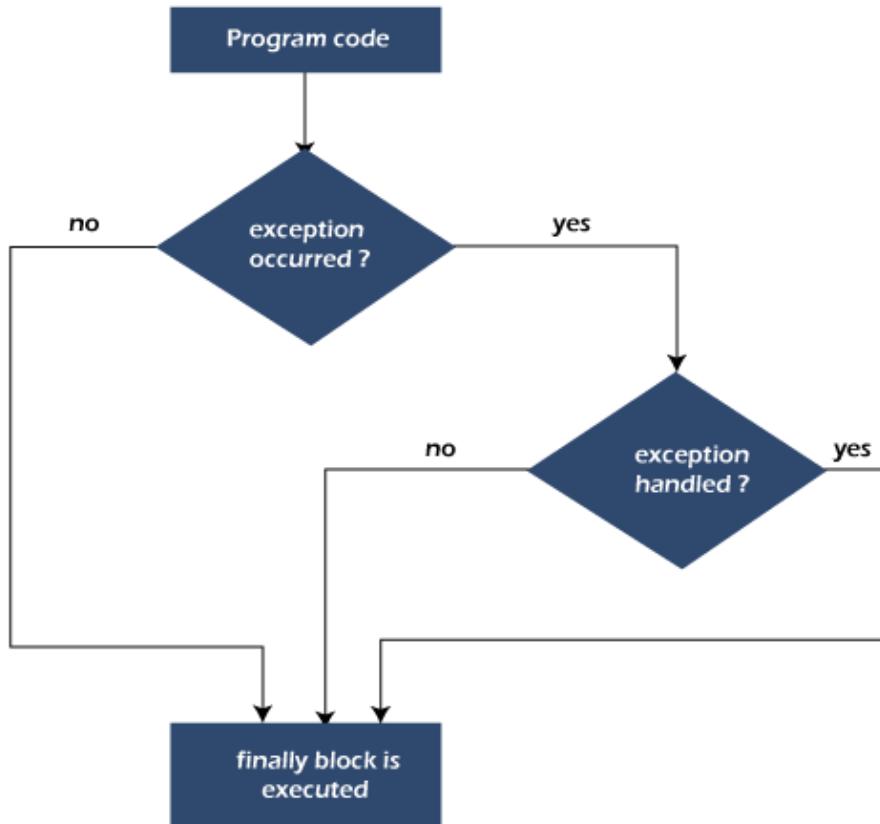
Java finally block

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

```
class TestFinallyBlock {  
    public static void main(String args[]){  
        try{  
            //below code do not throw any exception  
            int data=25/5;  
            System.out.println(data);  
        }  
        //catch won't be executed  
        catch(NullPointerException e){  
            System.out.println(e);  
        }  
        //executed regardless of exception occurred or not  
        finally {  
            System.out.println("finally block is always executed");  
        }  
  
        System.out.println("rest of the code...");  
    }  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java  
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock  
5  
finally block is always executed  
rest of the code...
```

Case 2: When an exception occur but not handled by the catch block

Let's see the the following 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

```
public class TestFinallyBlock1{
    public static void main(String args[]){
        try {
            System.out.println("Inside the try block");

            //below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }

        //cannot handle Arithmetic type exception
        //can only accept Null Pointer type exception
        catch(NullPointerException e){
            System.out.println(e);
        }

        //executes regardless of exception occured or not
        finally {
            System.out.println("finally block is always executed");
        }

        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmetricException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

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

```
public class TestFinallyBlock2{
    public static void main(String args[]){
        try {
            System.out.println("Inside try block");

            //below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }

        //handles the Arithmetic Exception / Divide by zero exception
        catch(ArithmetricException e){
            System.out.println("Exception handled");
            System.out.println(e);
        }

        //executes regardless of exception occurred or not
        finally {
            System.out.println("finally block is always executed");
        }
    }
}
```

```
System.out.println("rest of the code...");  
}  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java  
C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2  
Inside try block  
Exception handled  
java.lang.ArithmetricException: / by zero  
finally block is always executed  
rest of the code...
```

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).

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

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.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
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.

```
public class TestThrow1 {
    //function to check if person is eligible to vote or not
    public static void validate(int age) {
        if(age<18) {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
        }
        else {
            System.out.println("Person is eligible to vote!!!");
        }
    }
    //main method
    public static void main(String args[]){
        //calling the function
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

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

```
import java.io.*;

public class TestThrow2 {

    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {

        FileReader file = new FileReader("C:\\\\Users\\\\Anurati\\\\Desktop\\\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);

        throw new FileNotFoundException();
    }

    //main method
    public static void main(String args[]){
        try {
            method();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
    }
}
```

```
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrow2  
java.io.FileNotFoundException  
    at TestThrow2.method(TestThrow2.java:12)  
    at TestThrow2.main(TestThrow2.java:22)  
rest of the code...
```

Example 3: Throwing User-defined Exception

exception is everything else under the Throwable class.

TestThrow3.java

```
// class represents user-defined exception  
class UserDefinedException extends Exception  
{  
    public UserDefinedException(String str)  
    {  
        // Calling constructor of parent Exception  
        super(str);  
    }  
}  
// Class that uses above MyException  
public class TestThrow3  
{  
    public static void main(String args[])  
    {  
        try  
        {  
            // throw an object of user defined exception  
            throw new UserDefinedException("This is user-defined exception");  
        }  
        catch (UserDefinedException ude)
```

```
{  
    System.out.println("Caught the exception");  
    // Print the message from MyException object  
    System.out.println(ude.getMessage());  
}  
}  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow3.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrow3  
Caught the exception  
This is user-defined exception
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

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

```
class TestExceptionPropagation1{
    void m(){
        int data=50/0;
    }
    void n(){
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        TestExceptionPropagation1 obj=new TestExceptionPropagation1();
        obj.p();
        System.out.println("normal flow... ");
    }
}
```

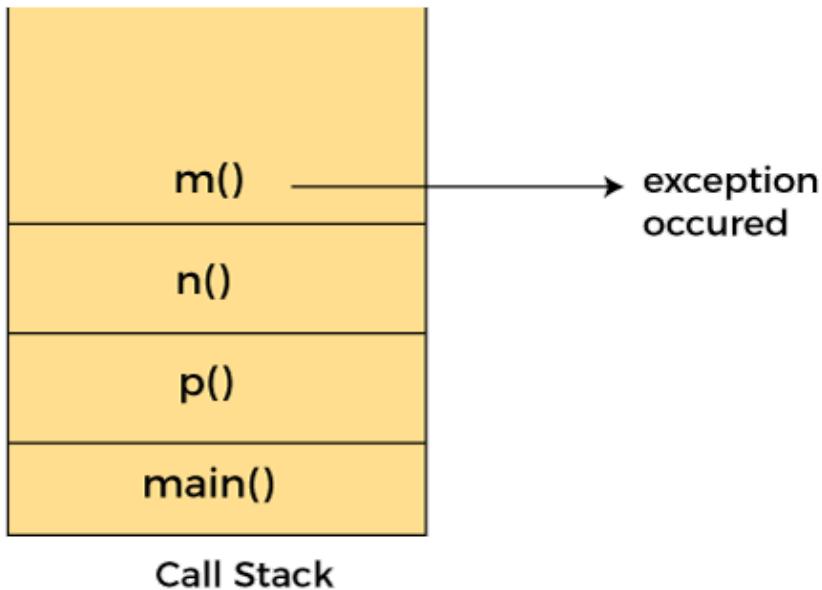
[Test it Now](#)

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.

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

```
class TestExceptionPropagation2{
    void m(){
        throw new java.io.IOException("device error");//checked exception
    }
    void n(){
        m();
    }
    void p(){
        try{
```

```
n();  
}catch(Exception e){System.out.println("exception handled");}  
}  
public static void main(String args[]){  
TestExceptionPropagation2 obj=new TestExceptionPropagation2();  
obj.p();  
System.out.println("normal flow");  
}  
}
```

Test it Now

Output:

Compile Time Error

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

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

```
return_type method_name() throws exception_class_name{  
    //method code  
}
```

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).

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

```
import java.io.IOException;  
class Testthrows1{
```

```
void m()throws IOException{
    throw new IOException("device error");//checked exception
}

void n()throws IOException{
    m();
}

void p(){
try{
    n();
}catch(Exception e){System.out.println("exception handled");}
}

public static void main(String args[]){
    Testthrows1 obj=new Testthrows1();
    obj.p();
    System.out.println("normal flow...");
}
}
```

Test it Now

Output:

```
exception handled
normal flow...
```

Rule: If we are calling a method that declares an exception, we must either catch 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

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}

public class Testthrows2{
    public static void main(String args[]){
        try{
            M m=new M();
            m.method();
        }catch(Exception e){System.out.println("exception handled");}
        System.out.println("normal flow...");
    }
}
```

Test it Now

Output:

```
exception handled
    normal flow...
```

Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.



Let's see examples for both the scenario.

A) If exception does not occur

Testthrows3.java

```
import java.io.*;
class M{
    void method()throws IOException{
        System.out.println("device operation performed");
    }
}
class Testthrows3{
    public static void main(String args[])throws IOException{//declare exception
        M m=new M();
        m.method();

        System.out.println("normal flow...");
    }
}
```

Test it Now

Output:

```
device operation performed
normal flow...
```

B) If exception occurs

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Testthrows4.java

```
import java.io.*;
class M{
    void method()throws IOException{
        throw new IOException("device error");
    }
}
class Testthrows4{
```

```
public static void main(String args[])throws IOException{//declare exception
    M m=new M();
    m.method();

    System.out.println("normal flow...");
}
}
```

[Test it Now](#)

Output:

```
Exception in thread "main" java.io.IOException: device error
at M.method(Testthrows4.java:4)
at Testthrows4.main(Testthrows4.java:10)
```

Difference between throw and throws

[Click me for details](#)

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

← Prev

Next →

AD

 [Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Difference between throw and throws in Java

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

There are many differences between **throw** and **throws** keywords. A list of differences between throw and throws are given below:

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Java throw Example

TestThrow.java

```
public class TestThrow {
    //defining a method
    public static void checkNum(int num) {
        if (num < 1) {
            throw new ArithmeticException("\nNumber is negative, cannot calculate square");
        }
        else {
            System.out.println("Square of " + num + " is " + (num*num));
        }
    }
    //main method
    public static void main(String[] args) {
        TestThrow obj = new TestThrow();
        obj.checkNum(-3);
        System.out.println("Rest of the code..");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmetricException:
Number is negative, cannot calculate square
    at TestThrow.checkNum(TestThrow.java:6)
    at TestThrow.main(TestThrow.java:16)
```

Java throws Example

TestThrows.java

```
public class TestThrows {
    //defining a method
    public static int divideNum(int m, int n) throws ArithmetricException {
        int div = m / n;
```

```

return div;
}

//main method

public static void main(String[] args) {
    TestThrows obj = new TestThrows();
    try {
        System.out.println(obj.divideNum(45, 0));
    }
    catch (ArithmaticException e){
        System.out.println("\nNumber cannot be divided by 0");
    }

    System.out.println("Rest of the code..");
}
}

```

Output:

```

C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java
C:\Users\Anurati\Desktop\abcDemo>java TestThrows
Number cannot be divided by 0
Rest of the code..

```

Java throw and throws Example

TestThrowAndThrows.java

```

public class TestThrowAndThrows
{
    // defining a user-defined method
    // which throws ArithmaticException
    static void method() throws ArithmaticException
    {
        System.out.println("Inside the method()");
        throw new ArithmaticException("throwing ArithmaticException");
    }
    //main method
    public static void main(String args[])
}

```

```
{  
try  
{  
    method();  
}  
catch(ArithmeticException e)  
{  
    System.out.println("caught in main() method");  
}  
}  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java  
C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows  
Inside the method()  
caught in main() method
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and variables.	Finally block is always related to the try and catch block in exception handling.	finalize() method is used with the objects.
3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.

4.	Execution	Final method is executed only when we call it.	Finally block is executed as soon as the try-catch block is executed. It's execution is not dependant on the exception.	finalize method is executed just before the object is destroyed.
----	-----------	--	--	--

Java final Example

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

FinalExampleTest.java

```
public class FinalExampleTest {
    //declaring final variable
    final int age = 18;
    void display() {

        // reassigning value to age variable
        // gives compile time error
        age = 55;
    }

    public static void main(String[] args) {

        FinalExampleTest obj = new FinalExampleTest();
        // gives compile time error
        obj.display();
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
    age = 55;
          ^
1 error
```

In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

Java finally Example

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

FinallyExample.java

```
public class FinallyExample {
    public static void main(String args[]){
        try {
            System.out.println("Inside try block");
            // below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }
        // handles the Arithmetic Exception / Divide by zero exception
        catch (ArithmaticException e){
            System.out.println("Exception handled");
            System.out.println(e);
        }
        // executes regardless of exception occurred or not
        finally {
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmetricException: / by zero
finally block is always executed
rest of the code...
```

Java finalize Example

FinalizeExample.java

```
public class FinalizeExample {
    public static void main(String[] args) {
        FinalizeExample obj = new FinalizeExample();
        // printing the hashCode
        System.out.println("HashCode is: " + obj.hashCode());
        obj = null;
        // calling the garbage collector using gc()
        System.gc();
        System.out.println("End of the garbage collection");
    }
    // defining the finalize method
    protected void finalize()
    {
        System.out.println("Called the finalize() method");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java
Note: FinalizeExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample
HashCode is: 746292446
End of the garbage collection
Called the finalize() method
```

Exception Handling with Method Overriding in Java

There are many rules if we talk about method overriding with exception handling.

Some of the rules are listed below:

- **If the superclass method does not declare an exception**
 - If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
- **If the superclass method declares an exception**
 - If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

If the superclass method does not declare an exception

Rule 1: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.

Let's consider following example based on the above rule.

TestExceptionChild.java

```
import java.io.*;  
class Parent{  
  
    // defining the method  
    void msg() {  
        System.out.println("parent method");  
    }  
}  
  
public class TestExceptionChild extends Parent{  
  
    // overriding the method in child class  
    // gives compile time error  
    void msg() throws IOException {  
        System.out.println("TestExceptionChild");  
    }  
}
```

```
}
```



```
public static void main(String args[]) {  
    Parent p = new TestExceptionChild();  
    p.msg();  
}  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild.java  
TestExceptionChild.java:14: error: msg() in TestExceptionChild cannot override msg()  
) in Parent  
    void msg() throws IOException {  
        ^  
    overridden method does not throw IOException  
1 error
```

Rule 2: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.

TestExceptionChild1.java

```
import java.io.*;  
  
class Parent{  
    void msg() {  
        System.out.println("parent method");  
    }  
}  
  
class TestExceptionChild1 extends Parent{  
    void msg()throws ArithmeticException {  
        System.out.println("child method");  
    }  
  
    public static void main(String args[]) {  
        Parent p = new TestExceptionChild1();  
        p.msg();  
    }  
}
```

```
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild1.java
C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild1
child method
```

If the superclass method declares an exception

Rule 1: If the superclass method declares an exception, subclass overridden method can declare the same subclass exception or no exception but cannot declare parent exception.

Example in case subclass overridden method declares parent exception

TestExceptionChild2.java

```
import java.io.*;
class Parent{
    void msg()throws ArithmeticException {
        System.out.println("parent method");
    }
}

public class TestExceptionChild2 extends Parent{
    void msg()throws Exception {
        System.out.println("child method");
    }

    public static void main(String args[]) {
        Parent p = new TestExceptionChild2();

        try {
            p.msg();
        }
        catch (Exception e){}
    }
}
```

```
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild2.java
TestExceptionChild2.java:9: error: msg() in TestExceptionChild2 cannot override msg()
() in Parent
    void msg()throws Exception {
           ^
    overridden method does not throw Exception
1 error
```

Example in case subclass overridden method declares same exception

TestExceptionChild3.java

```
import java.io.*;
class Parent{
    void msg() throws Exception {
        System.out.println("parent method");
    }
}

public class TestExceptionChild3 extends Parent {
    void msg()throws Exception {
        System.out.println("child method");
    }
}

public static void main(String args[]){
    Parent p = new TestExceptionChild3();

    try {
        p.msg();
    }
    catch(Exception e) {}
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild3.java  
C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild3  
child method
```

Example in case subclass overridden method declares subclass exception

TestExceptionChild4.java

```
import java.io.*;  
  
class Parent{  
    void msg()throws Exception {  
        System.out.println("parent method");  
    }  
}  
  
class TestExceptionChild4 extends Parent{  
    void msg()throws ArithmeticException {  
        System.out.println("child method");  
    }  
  
    public static void main(String args[]){  
        Parent p = new TestExceptionChild4();  
  
        try {  
            p.msg();  
        }  
        catch(Exception e) {}  
    }  
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild4.java  
C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild4  
child method
```

Example in case subclass overridden method declares no exception

TestExceptionChild5.java

```
import java.io.*;
class Parent {
    void msg()throws Exception{
        System.out.println("parent method");
    }
}

class TestExceptionChild5 extends Parent{
    void msg() {
        System.out.println("child method");
    }

    public static void main(String args[]){
        Parent p = new TestExceptionChild5();

        try {
            p.msg();
        }
        catch(Exception e) {}

    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestExceptionChild5.java
C:\Users\Anurati\Desktop\abcDemo>java TestExceptionChild5
child method
```

← Prev

Next →

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.

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`:

```
public class WrongFileNameException extends Exception {  
    public WrongFileNameException(String errorMessage) {  
        super(errorMessage);  
    }  
}
```

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

```
// class representing custom exception
class InvalidAgeException extends Exception
{
    public InvalidAgeException (String str)
    {
        // calling the constructor of parent Exception
        super(str);
    }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

    // method to check the age
    static void validate (int age) throws InvalidAgeException{
        if(age < 18){

            // throw an object of user defined exception
            throw new InvalidAgeException("age is not valid to vote");
        }
        else {
            System.out.println("welcome to vote");
        }
    }

    // main method
    public static void main(String args[])
    {
    }
}
```

```
try
{
    // calling the method
    validate(13);

}
catch (InvalidAgeException ex)
{
    System.out.println("Caught the exception");

    // printing the message from InvalidAgeException object
    System.out.println("Exception occurred: " + ex);

}

System.out.println("rest of the code...");
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java
C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occurred: InvalidAgeException: age is not valid to vote
rest of the code...
```

Example 2:

TestCustomException2.java

```
// class representing custom exception
class MyCustomException extends Exception
{

}

// class that uses custom exception MyCustomException
public class TestCustomException2
{
    // main method
}
```

```
public static void main(String args[])
{
    try
    {
        // throw an object of user defined exception
        throw new MyCustomException();
    }

    catch (MyCustomException ex)
    {
        System.out.println("Caught the exception");
        System.out.println(ex.getMessage());
    }

    System.out.println("rest of the code...");
}

}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException2.java
C:\Users\Anurati\Desktop\abcDemo>java TestCustomException2
Caught the exception
null
rest of the code...
```

← Prev

Next →

AD

Java Inner Classes (Nested Classes)

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{  
    //code  
    class Java_Inner_class{  
        //code  
    }  
}
```

Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class**, including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization:** It requires less code to write.

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Do You Know

- What is the internal code generated by the compiler for member inner class?
- What are the two ways to create an anonymous inner class?
- Can we access the non-final local variable inside the local inner class?
- How to access the static nested class?
- Can we define an interface within the class?
- Can we define a class within the interface?

Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
 1. Member inner class
 2. Anonymous inner class
 3. Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

Java Member Inner class

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

Syntax:

```
class Outer{  
    //code  
    class Inner{  
        //code  
    }  
}
```

Java Member Inner Class Example

In this example, we are creating a msg() method in the member inner class that is accessing the private data member of the outer class.

TestMemberOuter1.java

```
class TestMemberOuter1{  
    private int data=30;  
    class Inner{  
        void msg(){System.out.println("data is "+data);}  
    }  
    public static void main(String args[]){  
        TestMemberOuter1 obj=new TestMemberOuter1();  
        TestMemberOuter1.Inner in=obj.new Inner();  
        in.msg();  
    }  
}
```

[Test it Now](#)

Output:

```
data is 30
```

How to instantiate Member Inner class in Java?

An object or instance of a member's inner class always exists within an object of its outer class. The new operator is used to create the object of member inner class with slightly different syntax.

The general form of syntax to create an object of the member inner class is as follows:

Syntax:

```
OuterClassReference.new MemberInnerClassConstructor();
```

Example:

```
obj.new Inner();
```

Here, OuterClassReference is the reference of the outer class followed by a dot which is followed by the new operator.

Internal working of Java member inner class

The java compiler creates two class files in the case of the inner class. The class file name of the inner class is "Outer\$Inner". If you want to instantiate the inner class, you must have to create the instance of the outer class. In such a case, an instance of inner class is created inside the instance of the outer class.

Internal code generated by the compiler

The Java compiler creates a class file named Outer\$Inner in this case. The Member inner class has the reference of Outer class that is why it can access all the data members of Outer class including private.

```
import java.io.PrintStream;
class Outer$Inner
{
    final Outer this$0;
    Outer$Inner()
}
```

```
{ super();
    this$0 = Outer.this;
}

void msg()
{
    System.out.println((new StringBuilder()).append("data is ")
        .append(Outer.access$000(Outer.this)).toString());
}
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Java Anonymous inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

TestAnonymousInner.java

```
abstract class Person{  
    abstract void eat();  
}  
  
class TestAnonymousInnner{  
    public static void main(String args[]){  
        Person p=new Person(){  
            void eat(){System.out.println("nice fruits");}  
        };  
        p.eat();  
    }  
}
```

Test it Now

Output:

```
nice fruits
```

Internal working of given code

```
Person p=new Person(){
    void eat(){System.out.println("nice fruits");}
};
```

1. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner$1 extends Person
{
    TestAnonymousInner$1(){}
    void eat()
    {
        System.out.println("nice fruits");
    }
}
```

Java anonymous inner class example using interface

```
interface Eatable{
    void eat();
}

class TestAnonymousInner1{
    public static void main(String args[]){
        Eatable e=new Eatable(){
            public void eat(){System.out.println("nice fruits");}
        };
        e.eat();
    }
}
```

Test it Now

Output:

```
nice fruits
```

Internal working of given code

It performs two main tasks behind this code:

```
Eatable p=new Eatable(){  
    void eat(){System.out.println("nice fruits");}  
};
```

1. A class is created, but its name is decided by the compiler, which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p', a reference variable of the Eatable type.

Internal class generated by the compiler

```
import java.io.PrintStream;  
  
static class TestAnonymousInner1$1 implements Eatable  
{  
    TestAnonymousInner1$1()  
    void eat(){System.out.println("nice fruits");}  
}
```

← Prev

Next →

Java Local inner class

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

Locallnner1.java

```
public class locallnner1{  
    private int data=30;//instance variable  
    void display(){  
        class Local{  
            void msg(){System.out.println(data);}  
        }  
        Local l=new Local();  
        l.msg();  
    }  
    public static void main(String args[]){  
        locallnner1 obj=new locallnner1();  
        obj.display();  
    }  
}
```

[Test it Now](#)

Output:

```
30
```

Internal class generated by the compiler

In such a case, the compiler creates a class named Simple\$1Local that has the reference of the outer class.

```
import java.io.PrintStream;
class localInner1$Local
{
    final localInner1 this$0;
    localInner1$Local()
    {
        super();
        this$0 = Simple.this;
    }
    void msg()
    {
        System.out.println(localInner1.access$000(localInner1.this));
    }
}
```

Rule: Local variables can't be private, public, or protected.

Rules for Java Local Inner class

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

Example of local inner class with local variable

LocalInner2.java

```
class localInner2{
    private int data=30;//instance variable
```

```
void display(){  
    int value=50;//local variable must be final till jdk 1.7 only  
    class Local{  
        void msg(){System.out.println(value);}  
    }  
    Local l=new Local();  
    l.msg();  
}  
  
public static void main(String args[]){  
    localInner2 obj=new localInner2();  
    obj.display();  
}
```

Test it Now

Output:

50

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: Join Now

Java static nested class

A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of the outer class, including private.
- The static nested class cannot access non-static (instance) data members or

Java static nested class example with instance method

TestOuter1.java

```
class TestOuter1{  
    static int data=30;  
    static class Inner{  
        void msg(){System.out.println("data is "+data);}  
    }  
    public static void main(String args[]){  
        TestOuter1.Inner obj=new TestOuter1.Inner();  
        obj.msg();  
    }  
}
```

Test it Now

Output:

```
data is 30
```

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of the Outer class because the nested class is static and static properties, methods, or classes can be accessed without an object.

Internal class generated by the compiler

```
import java.io.PrintStream;  
static class TestOuter1$Inner
```

```
{  
TestOuter1$Inner(){  
void msg(){  
System.out.println((new StringBuilder()).append("data is ")  
.append(TestOuter1.data).toString());  
}  
}  
}
```

Java static nested class example with a static method

If you have the static member inside the static nested class, you don't need to create an instance of the static nested class.

TestOuter2.java

```
public class TestOuter2{  
static int data=30;  
static class Inner{  
static void msg(){System.out.println("data is "+data);}  
}  
public static void main(String args[]){  
TestOuter2.Inner.msg();//no need to create the instance of static nested class  
}  
}
```

Test it Now

Output:

```
data is 30
```

← Prev

Next →

Java Nested Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

Syntax of nested interface which is declared within the interface

```
interface interface_name{  
...  
    interface nested_interface_name{  
        ...  
    }  
}
```

Syntax of nested interface which is declared within the class

```
class class_name{  
...  
    interface nested_interface_name{  
        ...  
    }  
}
```

Example of nested interface which is declared within the interface

In this example, we will learn how to declare the nested interface and how we can access it.

TestNestedInterface1.java

```

interface Showable{
    void show();
}

interface Message{
    void msg();
}

class TestNestedInterface1 implements Showable.Message{
    public void msg(){System.out.println("Hello nested interface");}
}

public static void main(String args[]){
    Showable.Message message=new TestNestedInterface1()//upcasting here
    message.msg();
}

```

Test it Now

Output:

```
hello nested interface
```

[download the example of nested interface](#)

As you can see in the above example, we are accessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like the almirah inside the room; we cannot access the almirah directly because we must enter the room first. In the collection framework, the sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map, i.e., accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates a public and static interface as displayed below:

```

public static interface Showable$Message
{
    public abstract void msg();
}

```

Example of nested interface which is declared within the class

Let's see how we can define an interface inside the class and how we can access it.

TestNestedInterface2.java

```
class A{  
    interface Message{  
        void msg();  
    }  
}  
  
class TestNestedInterface2 implements A.Message{  
    public void msg(){System.out.println("Hello nested interface");}  
  
    public static void main(String args[]){  
        A.Message message=new TestNestedInterface2();//upcasting here  
        message.msg();  
    }  
}
```

Test it Now

Output:

```
hello nested interface
```

Can we define a class inside the interface?

Yes, if we define a class inside the interface, the Java compiler creates a static nested class. Let's see how can we define a class within the interface:

```
interface M{  
    class A{}  
}
```

Multithreading in Java

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- 2) You **can perform many operations together, so it saves time**.
- 3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- o Process-based Multitasking (Multiprocessing)
- o Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- o Each process has an address in memory. In other words, each process allocates a separate memory area.
- o A process is heavyweight.
- o Cost of communication between the process is high.
- o Switching from one process to another requires some time for saving and loading **registers**, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

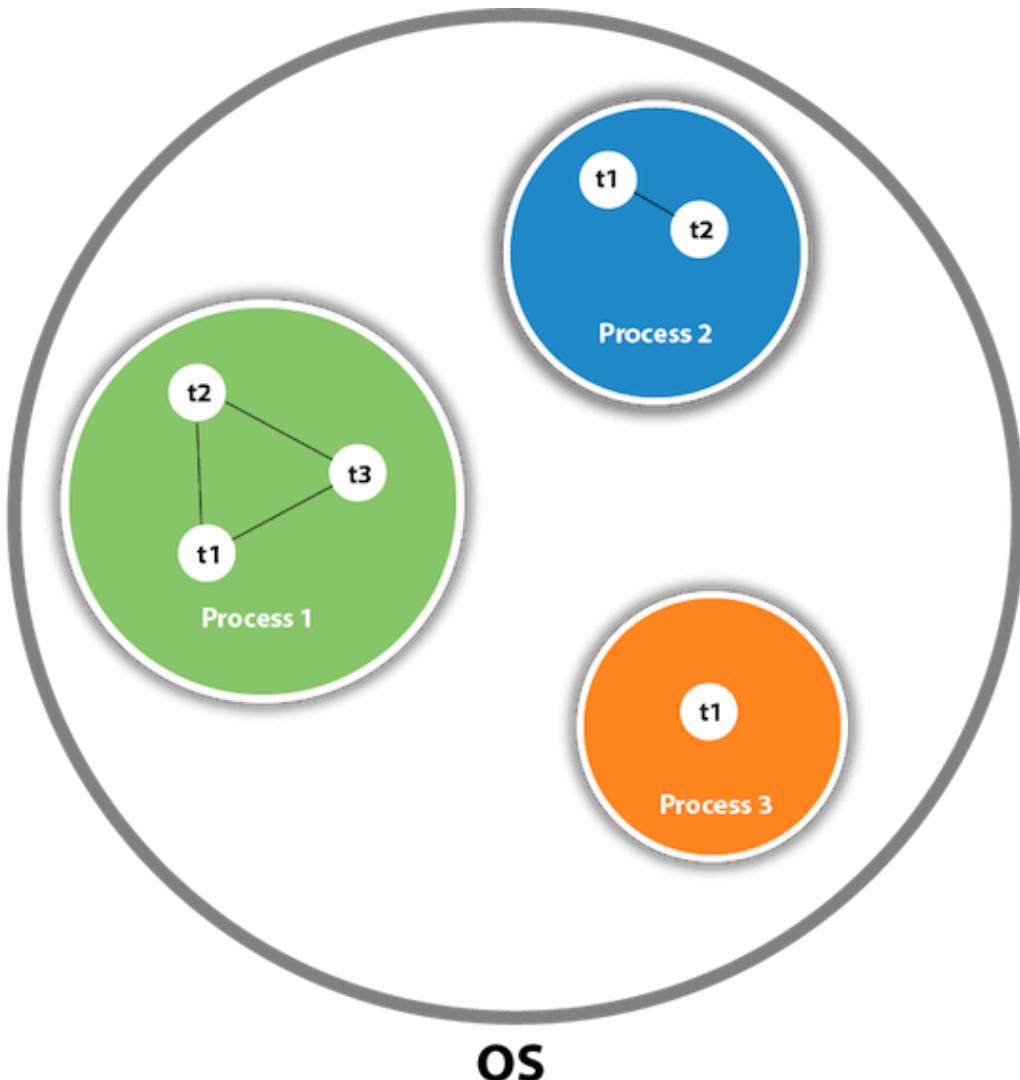
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides **constructors** and methods to create and perform operations on a thread. Thread class extends **Object class** and implements **Runnable interface**.

AD

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	<code>start()</code>	It is used to start the execution of the thread.
2)	void	<code>run()</code>	It is used to do an action for a thread.
3)	static void	<code>sleep()</code>	It sleeps a thread for the specified amount of time.
4)	static Thread	<code>currentThread()</code>	It returns a reference to the currently executing thread object.
5)	void	<code>join()</code>	It waits for a thread to die.
6)	int	<code>getPriority()</code>	It returns the priority of the thread.
7)	void	<code>setPriority()</code>	It changes the priority of the thread.
8)	String	<code>getName()</code>	It returns the name of the thread.
9)	void	<code>setName()</code>	It changes the name of the thread.

10)	long	<code>getId()</code>	It returns the id of the thread.
11)	boolean	<code>isAlive()</code>	It tests if the thread is alive.
12)	static void	<code>yield()</code>	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	<code>suspend()</code>	It is used to suspend the thread.
14)	void	<code>resume()</code>	It is used to resume the suspended thread.
15)	void	<code>stop()</code>	It is used to stop the thread.
16)	void	<code>destroy()</code>	It is used to destroy the thread group and all of its subgroups.
17)	boolean	<code>isDaemon()</code>	It tests if the thread is a daemon thread.

18)	void	<code>setDaemon()</code>	It marks the thread as daemon or user thread.
19)	void	<code>interrupt()</code>	It interrupts the thread.
20)	boolean	<code>isInterrupted()</code>	It tests whether the thread has been interrupted.
21)	static boolean	<code>interrupted()</code>	It tests whether the current thread has been interrupted.
22)	static int	<code>activeCount()</code>	It returns the number of active threads in the current thread's thread group.
23)	void	<code>checkAccess()</code>	It determines if the currently running thread has permission to modify the thread.

24)	static boolean	<code>holdLock()</code>	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	<code>dumpStack()</code>	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	<code>getStackTrace()</code>	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	<code>enumerate()</code>	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	<code>getState()</code>	It is used to return the state of the thread.

29)	ThreadGroup	<code>getThreadGroup()</code>	It is used to return the thread group to which this thread belongs
30)	String	<code>toString()</code>	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	<code>notify()</code>	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	<code>notifyAll()</code>	It is used to give the notification to all waiting threads of a particular object.
33)	void	<code>setContextClassLoader()</code>	It sets the context ClassLoader for the Thread.

34)	ClassLoader	<code>getContextClassLoader()</code>	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	<code>getDefaultUncaughtExceptionHandler()</code>	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	<code>setDefaultUncaughtExceptionHandler()</code>	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

Do You Know

- o How to perform two tasks by two threads?
- o How to perform multithreading by anonymous class?
- o What is the Thread Scheduler and what is the difference between preemptive scheduling and time slicing?
- o What happens if we start a thread twice?
- o What happens if we call the run() method instead of start() method?
- o What is the purpose of join method?
- o Why JVM terminates the daemon thread if no user threads are remaining?
- o What is the shutdown hook?
- o What is garbage collection?

- What is the purpose of finalize() method?
- What does the gc() method?
- What is synchronization and why use synchronization?
- What is the difference between synchronized method and synchronized block?
- What are the two ways to perform static synchronization?
- What is deadlock and when it can occur?
- What is interthread-communication or cooperation?

What will we learn in Multithreading

- Multithreading
- Life Cycle of a Thread
- Two ways to create a Thread
- How to perform multiple tasks by multiple threads
- Thread Scheduler
- Sleeping a thread
- Can we start a thread twice?
- What happens if we call the run() method instead of start() method?
- Joining a thread
- Naming a thread
- Priority of a thread
- Daemon Thread
- ShutdownHook
- Garbage collection
- Synchronization with synchronized method
- Synchronized block
- Static synchronization
- Deadlock
- Inter-thread communication

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

Explanation of Different Thread States

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **Runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread to the running state.

A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the `join()` method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the `sleep()` method on a specific thread. The `sleep()` method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

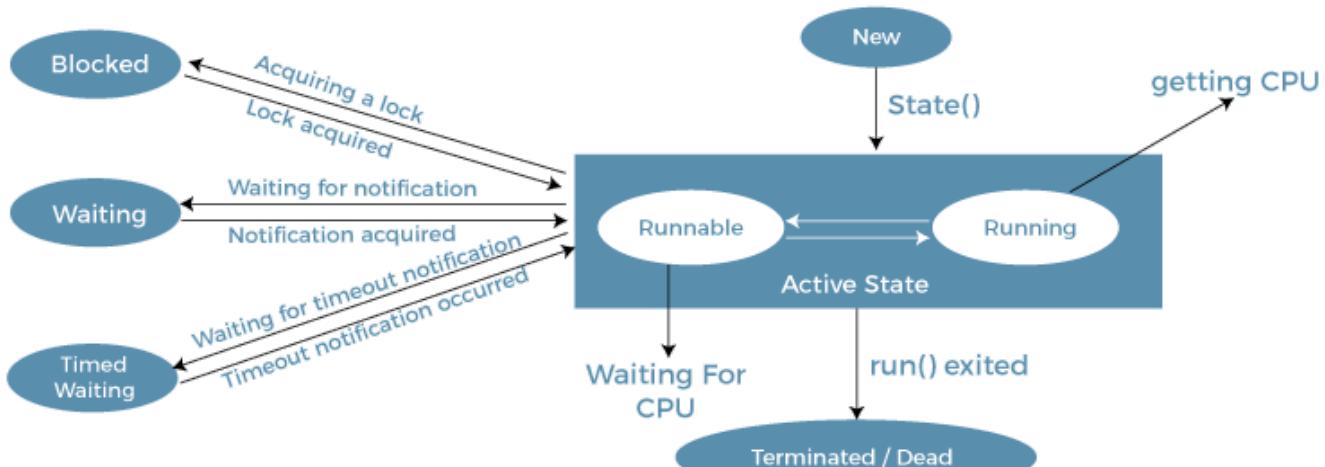
Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.



A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.



Life Cycle of a Thread

Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

public static final Thread.State NEW

It represents the first state of a thread that is the NEW state.

public static final Thread.State RUNNABLE

It represents the runnable state. It means a thread is waiting in the queue to run.

public static final Thread.State BLOCKED

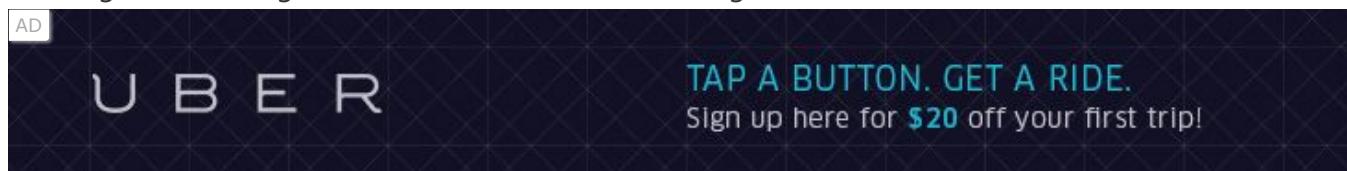
It represents the blocked state. In this state, the thread is waiting to acquire a lock.

public static final Thread.State WAITING

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

public static final Thread.State TIMED_WAITING

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.



- sleep
- join with timeout
- wait with timeout

- o parkUntil
- o parkNanos

```
public static final Thread.State TERMINATED
```

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Java Program for Demonstrating Thread States

The following Java program shows some of the states of a thread defined above.

FileName: ThreadState.java

```
// ABC class implements the interface Runnable
class ABC implements Runnable
{
    public void run()
    {

        // try-catch block
        try
        {
            // moving thread t2 to the state timed waiting
            Thread.sleep(100);
        }
        catch (InterruptedException ie)
        {
            ie.printStackTrace();
        }
    }
}
```

```
System.out.println("The state of thread t1 while it invoked the method join() on thread t2 - "
    "+ ThreadState.t1.getState());
```

```
// try-catch block
try
{
    Thread.sleep(200);
}
```

```
}

catch (InterruptedException ie)

{

ie.printStackTrace();

}

}

// ThreadState class implements the interface Runnable

public class ThreadState implements Runnable

{

public static Thread t1;

public static ThreadState obj;

// main method

public static void main(String argvs[])

{

// creating an object of the class ThreadState

obj = new ThreadState();

t1 = new Thread(obj);

// thread t1 is spawned

// The thread t1 is currently in the NEW state.

System.out.println("The state of thread t1 after spawning it - " + t1.getState());

// invoking the start() method on

// the thread t1

t1.start();

// thread t1 is moved to the Runnable state

System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getState());

}

public void run()

{

ABC myObj = new ABC();

Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.
```

```
System.out.println("The state of thread t2 after spawning it - " + t2.getState());
t2.start();

// thread t2 is moved to the runnable state
System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());

// try-catch block for the smooth flow of the program
try
{
// moving the thread t1 to the state timed waiting
Thread.sleep(200);
}

catch (InterruptedException ie)
{
ie.printStackTrace();
}

System.out.println("The state of thread t2 after invoking the method sleep() on it - " + t2.getState());

// try-catch block for the smooth flow of the program
try
{
// waiting for thread t2 to complete its execution
t2.join();
}

catch (InterruptedException ie)
{
ie.printStackTrace();
}

System.out.println("The state of thread t2 when it has completed its execution - " + t2.getState());
}
```

Output:

```
The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it - RUNNABLE
The state of thread t2 after spawning it - NEW
```

the state of thread t2 after calling the method start() on it - RUNNABLE

The state of thread t1 while it invoked the method join() on thread t2 -TIMED_WAITING

The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING

The state of thread t2 when it has completed it's execution - TERMINATED

Explanation: Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 get to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish it's execution as it has invoked the method join() on thread t2.

AD

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java Threads | How to create a thread in Java

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.

13. **public boolean isAlive()**: tests if the thread is alive.
14. **public void yield()**: causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend()**: is used to suspend the thread(deprecated).
16. **public void resume()**: is used to resume the suspended thread(deprecated).
17. **public void stop()**: is used to stop the thread(deprecated).
18. **public boolean isDaemon()**: tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b)**: marks the thread as daemon or user thread.
20. **public void interrupt()**: interrupts the thread.
21. **public boolean isInterrupted()**: tests if the thread has been interrupted.
22. **public static boolean interrupted()**: tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run()**: is used to perform action for a thread.

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.



1) Java Thread Example by extending Thread class

FileName: Multi.java

```
class Multi extends Thread{
    public void run(){
```

```

System.out.println("thread is running...");

}

public static void main(String args[]){
Multi t1=new Multi();
t1.start();
}
}

```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```

class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running... ");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1); // Using the constructor Thread(Runnable r)
t1.start();
}
}

```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```
public class MyThread1
{
    // Main method
    public static void main(String args[])
    {
        // creating an object of the Thread class using the constructor Thread(String name)
        Thread t= new Thread("My first thread");

        // the start() method moves the thread to the active state
        t.start();
        // getting the thread name by invoking the getName() method
        String str = t.getName();
        System.out.println(str);
    }
}
```

Output:

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

FileName: MyThread2.java

```
public class MyThread2 implements Runnable
{
    public void run()
    {
        System.out.println("Now the thread is running ...");
    }
}
```

```
// main method
public static void main(String args[])
{
    // creating an object of the class MyThread2
    Runnable r1 = new MyThread2();

    // creating an object of the class Thread using Thread(Runnable r, String name)
    Thread th1 = new Thread(r1, "My new thread");

    // the start() method moves the thread to the active state
    th1.start();

    // getting the thread name by invoking the getName() method
    String str = th1.getName();
    System.out.println(str);
}
```

Output:

```
My new thread
Now the thread is running ...
```

← Prev

Next →

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread Scheduler Algorithms

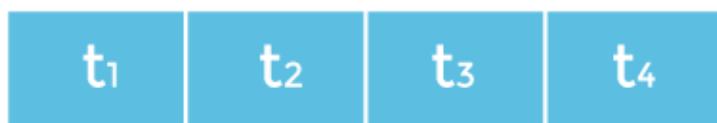
On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

First Come First Serve Scheduling:

In this scheduling algorithm, the scheduler picks the threads that arrive first in the runnable queue. Observe the following table:

Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3

In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.

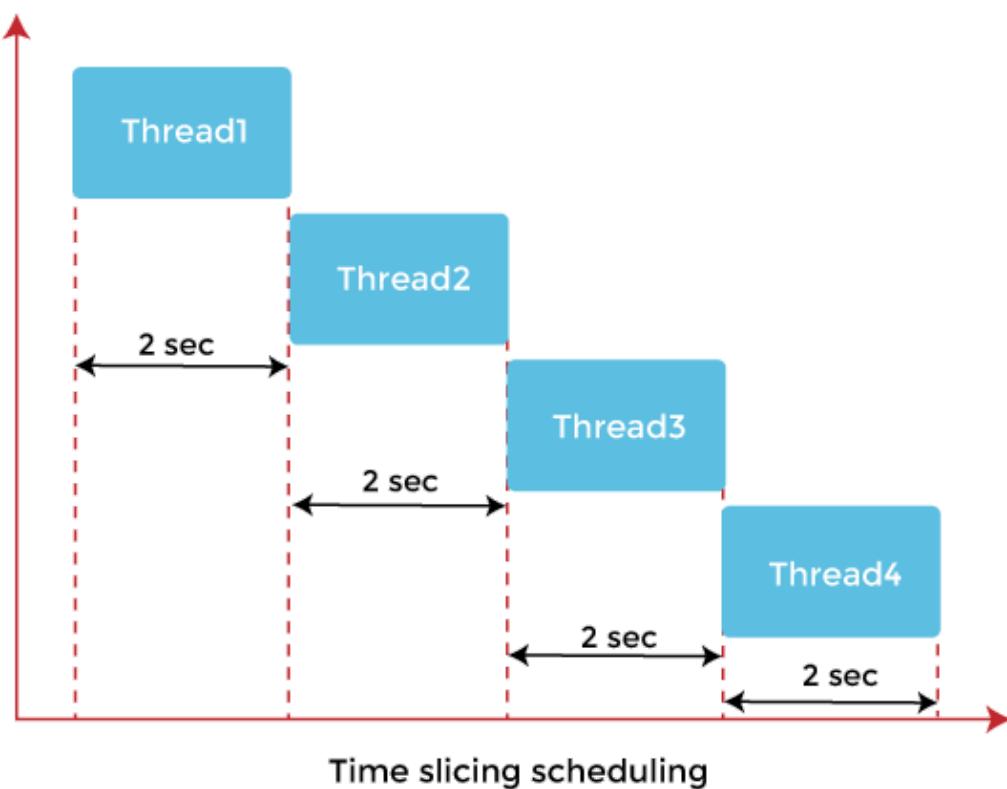


First Come First Serve Scheduling

Hence, Thread t₁ will be processed first, and Thread t₄ will be processed last.

Time-slicing scheduling:

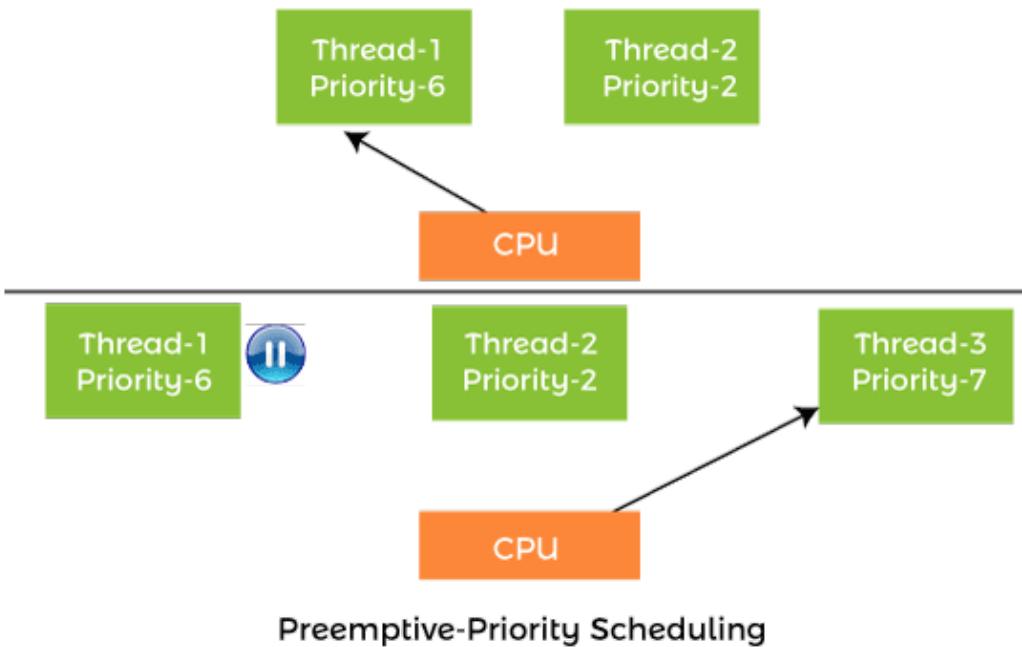
Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.



In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

Preemptive-Priority Scheduling:

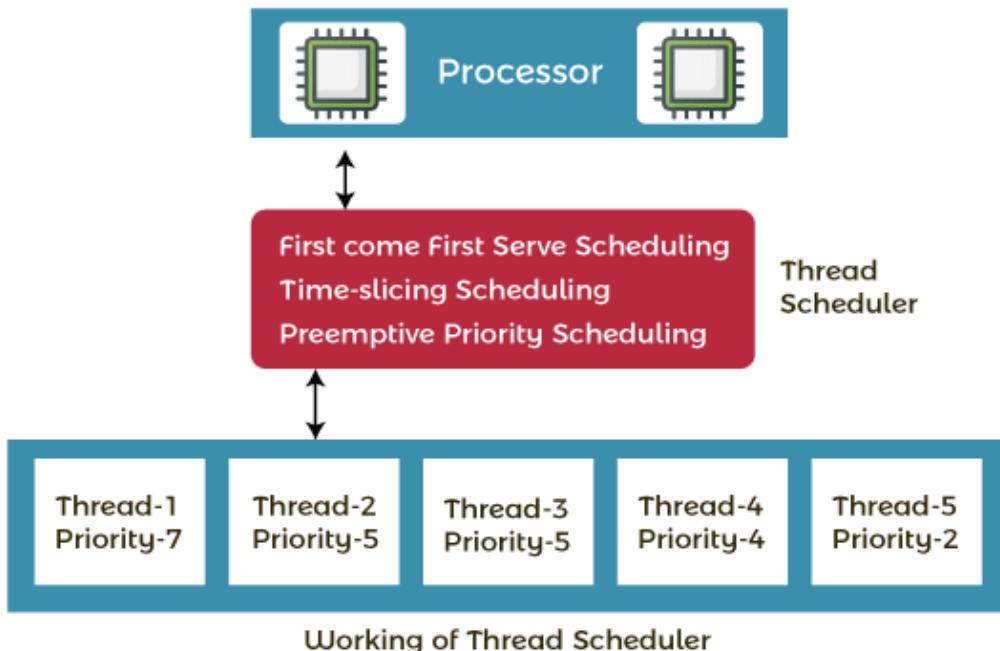
The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



Preemptive-Priority Scheduling

Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

Working of the Java Thread Scheduler



Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

 [SPSS tutorial](#)

Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

Following are the syntax of the sleep() method.

```
public static void sleep(long mls) throws InterruptedException  
public static void sleep(long mls, int n) throws InterruptedException
```

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters:

The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the `InterruptedException` is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the `sleep()` method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Example of the `sleep()` method in Java : on the custom thread

The following example shows how one can use the `sleep()` method on the custom thread.

FileName: TestSleepMethod1.java

```
class TestSleepMethod1 extends Thread{
    public void run(){
        for(int i=1;i<5;i++){
            // the thread will sleep for the 500 milli seconds
            try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }

    public static void main(String args[]){
        TestSleepMethod1 t1=new TestSleepMethod1();
        TestSleepMethod1 t2=new TestSleepMethod1();

        t1.start();
        t2.start();
    }
}
```

Output:

```
1
1
2
2
3
3
```

4

4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Example of the sleep() Method in Java : on the main thread

FileName: TestSleepMethod2.java

```
// important import statements
import java.lang.Thread;
import java.io.*;

public class TestSleepMethod2
{
    // main method
    public static void main(String argvs[])
    {

        try {
            for (int j = 0; j < 5; j++)
            {

                // The main thread sleeps for the 1000 milliseconds, which is 1 sec
                // whenever the loop runs
                Thread.sleep(1000);

                // displaying the value of the variable
                System.out.println(j);
            }
        }

        catch (Exception expn)
        {
            // catching the exception
            System.out.println(expn);
        }
    }
}
```

```
}
```

```
}
```

Output:

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

```
0  
1  
2  
3  
4
```

Example of the sleep() Method in Java: When the sleeping time is -ive

The following example throws the exception `IllegalArgumentException` when the time for sleeping is negative.

FileName: TestSleepMethod3.java

```
// important import statements  
import java.lang.Thread;  
import java.io.*;  
  
public class TestSleepMethod3  
{  
    // main method  
    public static void main(String args[])  
    {  
        // we can also use throws keyword followed by  
        // exception name for throwing the exception  
        try  
        {  
            for (int j = 0; j < 5; j++)  
            {  
                // it throws the exception IllegalArgumentException
```

```
// as the time is -ive which is -100
```

```
Thread.sleep(-100);
```

```
// displaying the variable's value
```

```
System.out.println(j);
```

```
}
```

```
}
```

```
catch (Exception expn)
```

```
{
```

```
// the exception is caught here
```

```
System.out.println(expn);
```

```
}
```

```
}
```

```
}
```

Output:

```
java.lang.IllegalArgumentException: timeout value is negative
```

← Prev

Next →

AD



For Videos Join Our Youtube Channel: [Join Now](#)

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
public class TestThreadTwice1 extends Thread{
    public void run(){
        System.out.println("running...");
    }
    public static void main(String args[]){
        TestThreadTwice1 t1=new TestThreadTwice1();
        t1.start();
        t1.start();
    }
}
```

Test it Now

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

← Prev

Next →

What if we call Java run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

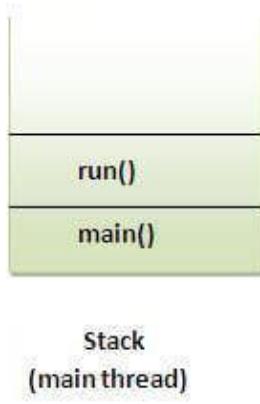
FileName: TestCallRun1.java

```
class TestCallRun1 extends Thread{  
    public void run(){  
        System.out.println("running...");  
    }  
    public static void main(String args[]){  
        TestCallRun1 t1=new TestCallRun1();  
        t1.run(); //fine, but does not start a separate call stack  
    }  
}
```

Test it Now

Output:

```
running...
```



Problem if you direct call run() method

FileName: TestCallRun2.java

```
class TestCallRun2 extends Thread{  
    public void run(){  
        for(int i=1;i<5;i++){  
            try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}  
            System.out.println(i);  
        }  
    }  
  
    public static void main(String args[]){  
        TestCallRun2 t1=new TestCallRun2();  
        TestCallRun2 t2=new TestCallRun2();  
  
        t1.run();  
        t2.run();  
    }  
}
```

Test it Now**Output:**

```
1  
2  
3  
4  
1  
2  
3  
4
```

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

← Prev

Next →

Java join() method

The join() method in Java is provided by the `java.lang.Thread` class that permits one thread to wait until the other thread to finish its execution. Suppose `th` be the object the class `Thread` whose thread is doing its execution currently, then the `th.join();` statement ensures that `th` is finished before the program does the execution of the next statement. When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period. However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters. The following are the three overloaded join() methods.

Description of The Overloaded join() Method

join(): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the `InterruptedException`.

Syntax:

```
public final void join() throws InterruptedException
```

join(long mls): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds) is over.

Syntax:

```
public final synchronized void join(long mls) throws InterruptedException, where mls is in millise
```

join(long mls, int nanos): When the join() method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

```
public final synchronized void join(long mls, int nanos) throws InterruptedException, where mls i
```

Example of join() Method in Java

The following program shows the usage of the join() method.

FileName: ThreadJoinExample.java

```
// A Java program for understanding
// the joining of threads

// import statement
import java.io.*;

// The ThreadJoin class is the child class of the class Thread
class ThreadJoin extends Thread
{
    // overriding the run method
    public void run()
    {
        for (int j = 0; j < 2; j++)
        {
            try
            {
                // sleeping the thread for 300 milli seconds
                Thread.sleep(300);
                System.out.println("The current thread name is: " + Thread.currentThread().getName());
            }
            // catch block for catching the raised exception
            catch(Exception e)
            {
                System.out.println("The exception has been caught: " + e);
            }
            System.out.println(j );
        }
    }
}
```

```
public class ThreadJoinExample
{
    // main method
    public static void main (String argvs[])
    {

        // creating 3 threads
        ThreadJoin th1 = new ThreadJoin();
        ThreadJoin th2 = new ThreadJoin();
        ThreadJoin th3 = new ThreadJoin();

        // thread th1 starts
        th1.start();

        // starting the second thread after when
        // the first thread th1 has ended or died.

        try
        {
            System.out.println("The current thread name is: " + Thread.currentThread().getName());

            // invoking the join() method
            th1.join();
        }

        // catch block for catching the raised exception
        catch(Exception e)
        {
            System.out.println("The exception has been caught " + e);
        }

        // thread th2 starts
        th2.start();

        // starting the th3 thread after when the thread th2 has ended or died.

        try
        {
            System.out.println("The current thread name is: " + Thread.currentThread().getName());
        }
    }
}
```

```
th2.join();
}

// catch block for catching the raised exception
catch(Exception e)
{
    System.out.println("The exception has been caught " + e);
}

// thread th3 starts
th3.start();
}
}
```

Output:

```
The current thread name is: main
The current thread name is: Thread - 0
0
The current thread name is: Thread - 0
1
The current thread name is: main
The current thread name is: Thread - 1
0
The current thread name is: Thread - 1
1
The current thread name is: Thread - 2
0
The current thread name is: Thread - 2
1
```

Explanation: The above program shows that the second thread th2 begins after the first thread th1 has ended, and the thread th3 starts its work after the second thread th2 has ended or died.

The Join() Method: InterruptedException

We have learnt in the description of the join() method that whenever the interruption of the thread occurs, it leads to the throwing of InterruptedException. The following example shows the same.

FileName: ThreadJoinExample1.java

```
class ABC extends Thread
{
    Thread threadToInterrupt;
    // overriding the run() method
    public void run()
    {
        // invoking the method interrupt
        threadToInterrupt.interrupt();
    }
}

public class ThreadJoinExample1
{
    // main method
    public static void main(String[] args)
    {
        try
        {
            // creating an object of the class ABC
            ABC th1 = new ABC();

            th1.threadToInterrupt = Thread.currentThread();
            th1.start();

            // invoking the join() method leads
            // to the generation of InterruptedException
            th1.join();
        }
        catch (InterruptedException ex)
        {
            System.out.println("The exception has been caught. " + ex);
        }
    }
}
```

```
}
```

Output:

```
The exception has been caught. java.lang.InterruptedException
```

AD

Some More Examples of the join() Method

Let's see some other examples.

Filename: TestJoinMethod1.java

```
class TestJoinMethod1 extends Thread{
    public void run(){
        for(int i=1;i<=5;i++){
            try{
                Thread.sleep(500);
            }catch(Exception e){System.out.println(e);}
            System.out.println(i);
        }
    }

    public static void main(String args[]){
        TestJoinMethod1 t1=new TestJoinMethod1();
        TestJoinMethod1 t2=new TestJoinMethod1();
        TestJoinMethod1 t3=new TestJoinMethod1();
        t1.start();
        try{
            t1.join();
        }catch(Exception e){System.out.println(e);}

        t2.start();
        t3.start();
    }
}
```

}

Output:

AD

```
1  
2  
3  
4  
5  
1  
1  
2  
2  
3  
3  
4  
4  
5  
5
```

We can see in the above example, when t1 completes its task then t2 and t3 starts executing.

join(long milliseconds) Method Example

Filename: TestJoinMethod2.java

```
class TestJoinMethod2 extends Thread{  
    public void run(){  
        for(int i=1;i<=5;i++){  
            try{  
                Thread.sleep(500);  
            }catch(Exception e){System.out.println(e);}  
            System.out.println(i);  
        }  
    }  
}
```

```
public static void main(String args[]){
    TestJoinMethod2 t1=new TestJoinMethod2();
    TestJoinMethod2 t2=new TestJoinMethod2();
    TestJoinMethod2 t3=new TestJoinMethod2();
    t1.start();
    try{
        t1.join(1500);
    }catch(Exception e){System.out.println(e);}

    t2.start();
    t3.start();
}
```

Output:

```
1
2
3
1
4
1
2
5
2
3
3
4
4
5
5
```

In the above example, when t1 completes its task for 1500 milliseconds(3 times), then t2 and t3 start executing.

Naming Thread and Current Thread

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the setName() method. The syntax of setName() and getName() methods are given below:

public String getName(): is used to **return** the name of a thread.

public void setName(String name): is used to change the name of a thread.

We can also set the name of a thread directly when we create a new thread using the constructor of the class.

Example of naming a thread : Using setName() Method

FileName: TestMultiNaming1.java

```
class TestMultiNaming1 extends Thread{
    public void run(){
        System.out.println("running...");
    }
    public static void main(String args[]){
        TestMultiNaming1 t1=new TestMultiNaming1();
        TestMultiNaming1 t2=new TestMultiNaming1();
        System.out.println("Name of t1:"+t1.getName());
        System.out.println("Name of t2:"+t2.getName());

        t1.start();
        t2.start();

        t1.setName("Sonoo Jaiswal");
        System.out.println("After changing name of t1:"+t1.getName());
    }
}
```

Test it Now

Output:

```
Name of t1:Thread-0
Name of t2:Thread-1
After changing name of t1:Sonoo Jaiswal
running...
running...
```

Example of naming a thread : Without Using setName() Method

One can also set the name of a thread at the time of the creation of a thread, without using the `setName()` method. Observe the following code.

FileName: ThreadNamingExample.java

```
// A Java program that shows how one can
// set the name of a thread at the time
// of creation of the thread

// import statement
import java.io.*;

// The ThreadNameClass is the child class of the class Thread
class ThreadName extends Thread
{
    // constructor of the class
    ThreadName(String threadName)
    {
        // invoking the constructor of
        // the superclass, which is Thread class.
        super(threadName);
    }

    // overriding the method run()
    public void run()
    {
        System.out.println(" The thread is executing....");
    }
}
```

```
}

}

public class ThreadNamingExample
{
    // main method
    public static void main (String args[])
    {
        // creating two threads and setting their name
        // using the constructor of the class
        ThreadName th1 = new ThreadName("JavaTpoint1");
        ThreadName th2 = new ThreadName("JavaTpoint2");

        // invoking the getName() method to get the names
        // of the thread created above
        System.out.println("Thread - 1: " + th1.getName());
        System.out.println("Thread - 2: " + th2.getName());

        // invoking the start() method on both the threads
        th1.start();
        th2.start();
    }
}
```

Output:

```
Thread - 1: JavaTpoint1
Thread - 2: JavaTpoint2
The thread is executing....
The thread is executing....
```

Current Thread

The `currentThread()` method returns a reference of the currently executing thread.

```
public static Thread currentThread()
```

Example of currentThread() method

FileName: TestMultiNaming2.java

```
class TestMultiNaming2 extends Thread{
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String args[]){
        TestMultiNaming2 t1=new TestMultiNaming2();
        TestMultiNaming2 t2=new TestMultiNaming2();

        t1.start();
        t2.start();
    }
}
```

Test it Now

Output:

```
Thread-0
Thread-1
```

← Prev

Next →

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The `java.lang.Thread.getPriority()` method returns the priority of the given thread.

public final void setPriority(int newPriority): The `java.lang.Thread.setPriority()` method updates or assign the priority of the thread to `newPriority`. The method throws `IllegalArgumentException` if the value `newPriority` goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

Default priority of a thread is 5 (`NORM_PRIORITY`). The value of `MIN_PRIORITY` is 1 and the value of `MAX_PRIORITY` is 10.

Example of priority of a Thread:

FileName: ThreadPriorityExample.java

```
// Importing the required classes
import java.lang.*;

public class ThreadPriorityExample extends Thread
{
    // Method 1
}
```

```
// Whenever the start() method is called by a thread
// the run() method is invoked

public void run()
{
    // the print statement
    System.out.println("Inside the run() method");
}

// the main method
public static void main(String args[])
{
    // Creating threads with the help of ThreadPriorityExample class
    ThreadPriorityExample th1 = new ThreadPriorityExample();
    ThreadPriorityExample th2 = new ThreadPriorityExample();
    ThreadPriorityExample th3 = new ThreadPriorityExample();

    // We did not mention the priority of the thread.
    // Therefore, the priorities of the thread is 5, the default value

    // 1st Thread
    // Displaying the priority of the thread
    // using the getPriority() method
    System.out.println("Priority of the thread th1 is : " + th1.getPriority());

    // 2nd Thread
    // Display the priority of the thread
    System.out.println("Priority of the thread th2 is : " + th2.getPriority());

    // 3rd Thread
    // // Display the priority of the thread
    System.out.println("Priority of the thread th2 is : " + th2.getPriority());

    // Setting priorities of above threads by
    // passing integer arguments
    th1.setPriority(6);
    th2.setPriority(3);
    th3.setPriority(9);
```

```
// 6  
System.out.println("Priority of the thread th1 is : " + th1.getPriority());  
  
// 3  
System.out.println("Priority of the thread th2 is : " + th2.getPriority());  
  
// 9  
System.out.println("Priority of the thread th3 is : " + th3.getPriority());  
  
// Main thread  
  
// Displaying name of the currently executing thread  
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());  
  
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());  
  
// Priority of the main thread is 10 now  
Thread.currentThread().setPriority(10);  
  
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());  
}  
}
```

Output:

```
Priority of the thread th1 is : 5  
Priority of the thread th2 is : 5  
Priority of the thread th2 is : 5  
Priority of the thread th1 is : 6  
Priority of the thread th2 is : 3  
Priority of the thread th3 is : 9  
Currently Executing The Thread : main  
Priority of the main thread is : 5  
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java

thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

FileName: ThreadPriorityExample1.java

```
// importing the java.lang package
import java.lang.*;

public class ThreadPriorityExample1 extends Thread
{
    // Method 1
    // Whenever the start() method is called by a thread
    // the run() method is invoked
    public void run()
    {
        // the print statement
        System.out.println("Inside the run() method");
    }

    // the main method
    public static void main(String args[])
    {
        // Now, priority of the main thread is set to 7
        Thread.currentThread().setPriority(7);

        // the current thread is retrieved
        // using the currentThread() method

        // displaying the main thread priority
        // using the getPriority() method of the Thread class
        System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

        // creating a thread by creating an object of the class ThreadPriorityExample1
        ThreadPriorityExample1 th1 = new ThreadPriorityExample1();
    }
}
```

```
// th1 thread is the child of the main thread  
// therefore, the th1 thread also gets the priority 7  
  
// Displaying the priority of the current thread  
System.out.println("Priority of the thread th1 is : " + th1.getPriority());  
}  
}
```

Output:

```
Priority of the main thread is : 7  
Priority of the thread th1 is : 7
```

Explanation: If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

Example of IllegalArgumentException

We know that if the value of the parameter *newPriority* of the method *getPriority()* goes out of the range (1 to 10), then we get the *IllegalArgumentException*. Let's observe the same with the help of an example.

FileName: *IllegalArgumentException.java*

```
// importing the java.lang package  
import java.lang.*;  
  
public class IllegalArgumentException extends Thread  
{  
  
    // the main method  
    public static void main(String args[])  
    {  
  
        // Now, priority of the main thread is set to 17, which is greater than 10  
        Thread.currentThread().setPriority(17);  
    }  
}
```

```
// The current thread is retrieved  
// using the currentThread() method  
  
// displaying the main thread priority  
// using the getPriority() method of the Thread class  
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());  
  
}  
}
```

When we execute the above program, we get the following exception:

```
Exception in thread "main" java.lang.IllegalArgumentException  
    at java.base/java.lang.Thread.setPriority(Thread.java:1141)  
    at IllegalArgumentException.main(IllegalArgumentException.java:12)
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Daemon Thread in Java

Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The `java.lang.Thread` class provides two methods for java daemon thread.

No.	Method	Description
1)	<code>public void setDaemon(boolean status)</code>	is used to mark the current thread as daemon thread or user thread.
2)	<code>public boolean isDaemon()</code>	is used to check that current is daemon.

Simple example of Daemon thread in java

File: `MyThread.java`

```
public class TestDaemonThread1 extends Thread{
    public void run(){
```

```

if(Thread.currentThread().isDaemon())//checking for daemon thread
System.out.println("daemon thread work");
}
else{
System.out.println("user thread work");
}
}

public static void main(String[] args){
TestDaemonThread1 t1=new TestDaemonThread1()//creating thread
TestDaemonThread1 t2=new TestDaemonThread1();
TestDaemonThread1 t3=new TestDaemonThread1();

t1.setDaemon(true)//now t1 is daemon thread

t1.start()//starting threads
t2.start();
t3.start();
}
}

```

Test it Now

Output:

```

daemon thread work
user thread work
user thread work

```

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

File: MyThread.java

```

class TestDaemonThread2 extends Thread{
public void run(){
System.out.println("Name: "+Thread.currentThread().getName());
System.out.println("Daemon: "+Thread.currentThread().isDaemon());
}
}

```

```
}
```

```
public static void main(String[] args){  
    TestDaemonThread2 t1=new TestDaemonThread2();  
    TestDaemonThread2 t2=new TestDaemonThread2();  
    t1.start();  
    t1.setDaemon(true); //will throw exception here  
    t2.start();  
}  
}
```

Test it Now

Output:

```
exception in thread main: java.lang.IllegalThreadStateException
```

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.

In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, the thread is contained in the thread pool again.

Thread Pool Methods

newFixedThreadPool(int s): The method creates a thread pool of the fixed size s.

newCachedThreadPool(): The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.

newSingleThreadExecutor(): The method creates a new thread.

Advantage of Java Thread Pool

Better performance It saves time because there is no need to create a new thread.

Real time usage

It is used in Servlet and JSP where the container creates a thread pool to process the request.

Example of Java Thread Pool

Let's see a simple example of the Java thread pool using ExecutorService and Executors.

File: WorkerThread.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class WorkerThread implements Runnable {
    private String message;
    public WorkerThread(String s){
        this.message=s;
    }
    public void run() {
        System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
    }
}
```

```

processmessage();//call processmessage method that sleeps the thread for 2 seconds
System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
}

private void processmessage() {
    try { Thread.sleep(2000); } catch (InterruptedException e) { e.printStackTrace(); }
}
}

```

File: TestThreadPool.java

```

public class TestThreadPool {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread(""+ i);
            executor.execute(worker);//calling execute method of ExecutorService
        }
        executor.shutdown();
        while (!executor.isTerminated()) { }

        System.out.println("Finished all threads");
    }
}

```

Output:

```

pool-1-thread-1 (Start) message = 0
pool-1-thread-2 (Start) message = 1
pool-1-thread-3 (Start) message = 2
pool-1-thread-5 (Start) message = 4
pool-1-thread-4 (Start) message = 3
pool-1-thread-2 (End)
pool-1-thread-2 (Start) message = 5
pool-1-thread-1 (End)
pool-1-thread-1 (Start) message = 6
pool-1-thread-3 (End)
pool-1-thread-3 (Start) message = 7

```

```
pool-1-thread-4 (End)
pool-1-thread-4 (Start) message = 8
pool-1-thread-5 (End)
pool-1-thread-5 (Start) message = 9
pool-1-thread-2 (End)
pool-1-thread-1 (End)
pool-1-thread-4 (End)
pool-1-thread-3 (End)
pool-1-thread-5 (End)
Finished all threads
```

[download this example](#)

Thread Pool Example: 2

Let's see another example of the thread pool.

FileName: ThreadPoolExample.java

```
// important import statements
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.text.SimpleDateFormat;

class Tasks implements Runnable
{
    private String taskName;

    // constructor of the class Tasks
    public Tasks(String str)
    {
        // initializing the field taskName
        taskName = str;
    }

    // Printing the task name and then sleeps for 1 sec
    // The complete process is getting repeated five times
}
```

```
public void run()
{
try
{
for (int j = 0; j <= 5; j++)
{
if (j == 0)
{
Date dt = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");

//prints the initialization time for every task
System.out.println("Initialization time for the task name: " + taskName + " = " + sdf.format(dt));

}
else
{
Date dt = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");

// prints the execution time for every task
System.out.println("Time of execution for the task name: " + taskName + " = " + sdf.format(dt));

}

// 1000ms = 1 sec
Thread.sleep(1000);
}

System.out.println(taskName + " is complete.");
}

catch(InterruptedException ie)
{
ie.printStackTrace();
}
}
}
```

```
public class ThreadPoolExample
{
    // Maximum number of threads in the thread pool
    static final int MAX_TH = 3;

    // main method
    public static void main(String argvs[])
    {
        // Creating five new tasks
        Runnable rb1 = new Tasks("task 1");
        Runnable rb2 = new Tasks("task 2");
        Runnable rb3 = new Tasks("task 3");
        Runnable rb4 = new Tasks("task 4");
        Runnable rb5 = new Tasks("task 5");

        // creating a thread pool with MAX_TH number of
        // threads size the pool size is fixed
        ExecutorService pl = Executors.newFixedThreadPool(MAX_TH);

        // passes the Task objects to the pool to execute (Step 3)
        pl.execute(rb1);
        pl.execute(rb2);
        pl.execute(rb3);
        pl.execute(rb4);
        pl.execute(rb5);

        // pool is shutdown
        pl.shutdown();
    }
}
```

Output:

```
Initialization time for the task name: task 1 = 06 : 13 : 02
Initialization time for the task name: task 2 = 06 : 13 : 02
Initialization time for the task name: task 3 = 06 : 13 : 02
```

```
Time of execution for the task name: task 1 = 06 : 13 : 04
Time of execution for the task name: task 2 = 06 : 13 : 04
Time of execution for the task name: task 3 = 06 : 13 : 04
Time of execution for the task name: task 1 = 06 : 13 : 05
Time of execution for the task name: task 2 = 06 : 13 : 05
Time of execution for the task name: task 3 = 06 : 13 : 05
Time of execution for the task name: task 1 = 06 : 13 : 06
Time of execution for the task name: task 2 = 06 : 13 : 06
Time of execution for the task name: task 3 = 06 : 13 : 06
Time of execution for the task name: task 1 = 06 : 13 : 07
Time of execution for the task name: task 2 = 06 : 13 : 07
Time of execution for the task name: task 3 = 06 : 13 : 07
Time of execution for the task name: task 1 = 06 : 13 : 08
Time of execution for the task name: task 2 = 06 : 13 : 08
Time of execution for the task name: task 3 = 06 : 13 : 08
task 2 is complete.

Initialization time for the task name: task 4 = 06 : 13 : 09
task 1 is complete.

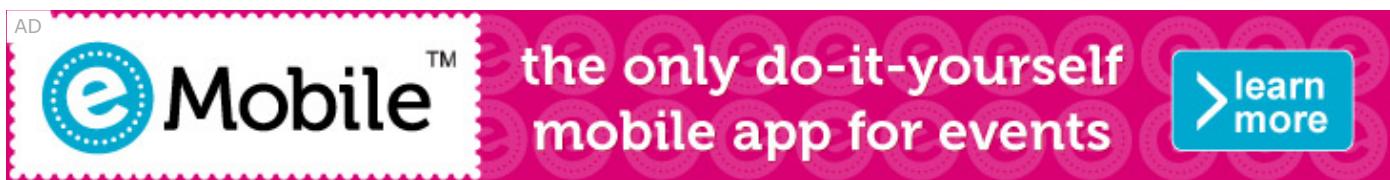
Initialization time for the task name: task 5 = 06 : 13 : 09
task 3 is complete.

Time of execution for the task name: task 4 = 06 : 13 : 10
Time of execution for the task name: task 5 = 06 : 13 : 10
Time of execution for the task name: task 4 = 06 : 13 : 11
Time of execution for the task name: task 5 = 06 : 13 : 11
Time of execution for the task name: task 4 = 06 : 13 : 12
Time of execution for the task name: task 5 = 06 : 13 : 12
Time of execution for the task name: task 4 = 06 : 13 : 13
Time of execution for the task name: task 5 = 06 : 13 : 13
Time of execution for the task name: task 4 = 06 : 13 : 14
Time of execution for the task name: task 5 = 06 : 13 : 14
task 4 is complete.

task 5 is complete.
```

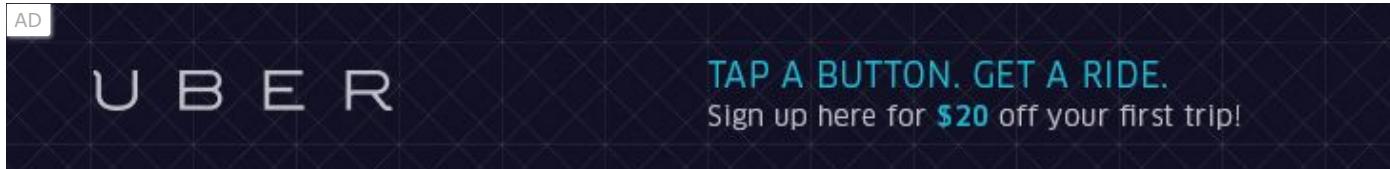
Explanation: It is evident by looking at the output of the program that tasks 4 and 5 are executed only when the thread has an idle thread. Until then, the extra tasks are put in the queue.

The takeaway from the above example is when one wants to execute 50 tasks but is not willing to create 50 threads. In such a case, one can create a pool of 10 threads. Thus, 10 out of 50 tasks are assigned, and the rest are put in the queue. Whenever any thread out of 10 threads becomes idle, it picks up the 11th task. The other pending tasks are treated the same way.



Risks involved in Thread Pools

The following are the risk involved in the thread pools.



Deadlock: It is a known fact that deadlock can come in any program that involves multithreading, and a thread pool introduces another scenario of deadlock. Consider a scenario where all the threads that are executing are waiting for the results from the threads that are blocked and waiting in the queue because of the non-availability of threads for the execution.

Thread Leakage: Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task. For example, when a thread throws the exception and the pool class is not able to catch this exception, then the thread exits and reduces the thread pool size by 1. If the same thing repeats a number of times, then there are fair chances that the pool will become empty, and hence, there are no threads available in the pool for executing other requests.

Resource Thrashing: A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

Points to Remember

Do not queue the tasks that are concurrently waiting for the results obtained from the other tasks. It may lead to a deadlock situation, as explained above.

Care must be taken whenever threads are used for the operation that is long-lived. It may result in the waiting of thread forever and will finally lead to the leakage of the resource.



In the end, the thread pool has to be ended explicitly. If it does not happen, then the program continues to execute, and it never ends. Invoke the shutdown() method on the thread pool to terminate the executor. Note that if someone tries to send another task to the executor after shutdown, it will throw a RejectedExecutionException.

One needs to understand the tasks to effectively tune the thread pool. If the given tasks are contrasting, then one should look for pools for executing different varieties of tasks so that one can properly tune them.

To reduce the probability of running JVM out of memory, one can control the maximum threads that can run in JVM. The thread pool cannot create new threads after it has reached the maximum limit.

A thread pool can use the same used thread if the thread has finished its execution. Thus, the time and resources used for the creation of a new thread are saved.

Tuning the Thread Pool

The accurate size of a thread pool is decided by the number of available processors and the type of tasks the threads have to execute. If a system has the P processors that have only got the computation type processes, then the maximum size of the thread pool of P or P + 1 achieves the maximum efficiency. However, the tasks may have to wait for I/O, and in such a scenario, one has to take into consideration the ratio of the waiting time (W) and the service time (S) for the request; resulting in the maximum size of the pool $P * (1 + W / S)$ for the maximum efficiency.

Conclusion

A thread pool is a very handy tool for organizing applications, especially on the server-side. Concept-wise, a thread pool is very easy to comprehend. However, one may have to look at a lot of issues when dealing with a thread pool. It is because the thread pool comes with some risks involved in it (risks are discussed above).

← Prev

Next →

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with a given parent group and name.

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods is given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.

3)	int	<code>activeGroupCount()</code>	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	void	<code>destroy()</code>	This method destroys the thread group and all of its subgroups.
5)	int	<code>enumerate(Thread[] list)</code>	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	int	<code>getMaxPriority()</code>	This method returns the maximum priority of the thread group.
7)	String	<code>getName()</code>	This method returns the name of the thread group.
8)	ThreadGroup	<code>getParent()</code>	This method returns the parent of the thread group.
9)	void	<code>interrupt()</code>	This method interrupts all threads in the thread group.
10)	boolean	<code>isDaemon()</code>	This method tests if the thread group is a daemon thread group.
11)	void	<code>setDaemon(boolean daemon)</code>	This method changes the daemon status of the thread group.
12)	boolean	<code>isDestroyed()</code>	This method tests if this thread group has been destroyed.
13)	void	<code>list()</code>	This method prints information about the thread group to the standard output.
14)	boolean	<code>parentOf(ThreadGroup g)</code>	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	<code>suspend()</code>	This method is used to suspend all threads in the thread group.
16)	void	<code>resume()</code>	This method is used to resume all threads in the thread group which was suspended using <code>suspend()</code> method.

17)	void	<code>setMaxPriority(int pri)</code>	This method sets the maximum priority of the group.
18)	void	<code>stop()</code>	This method is used to stop all threads in the thread group.
19)	String	<code>toString()</code>	This method returns a string representation of the Thread group.

Let's see a code to group multiple threads.

```
ThreadGroup tg1 = new ThreadGroup("Group A");
Thread t1 = new Thread(tg1,new MyRunnable(),"one");
Thread t2 = new Thread(tg1,new MyRunnable(),"two");
Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

ThreadGroup Example

File: *ThreadGroupDemo.java*

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

        Thread t1 = new Thread(tg1, runnable,"one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable,"two");
        t2.start();
    }
}
```

```

        Thread t3 = new Thread(tg1, runnable, "three");
        t3.start();

        System.out.println("Thread Group Name: " + tg1.getName());
        tg1.list();

    }
}

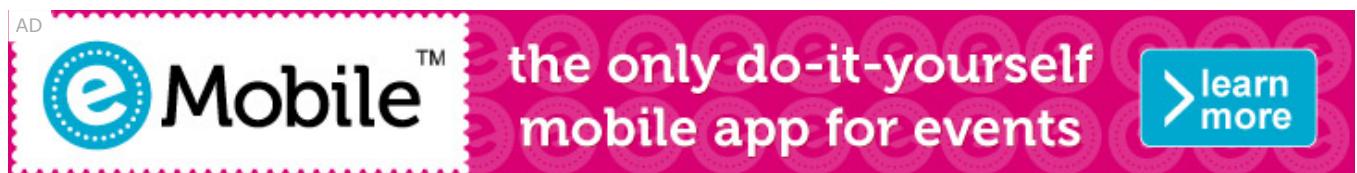
```

Output:

```

one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]

```



Thread Pool Methods Example: int activeCount()

Let's see how one can use the method activeCount().

FileName: ActiveCountExample.java

```
// code that illustrates the activeCount() method
```

```
// import statement
```

```
import java.lang.*;
```

```
class ThreadNew extends Thread
```

```
{
```

```
// constructor of the class
```

```
ThreadNew(String tName, ThreadGroup tgrp)
```

```
{
```

```
super(tgrp, tName);
```

```
start();  
}  
  
// overriding the run method  
public void run()  
{  
  
for (int j = 0; j < 1000; j++)  
{  
try  
{  
    Thread.sleep(5);  
}  
catch (InterruptedException e)  
{  
    System.out.println("The exception has been encountered " + e);  
}  
}  
}  
}  
}  
  
public class ActiveCountExample  
{  
// main method  
public static void main(String args[])  
{  
// creating the thread group  
ThreadGroup tg = new ThreadGroup("The parent group of threads");  
  
ThreadNew th1 = new ThreadNew("first", tg);  
System.out.println("Starting the first");  
  
ThreadNew th2 = new ThreadNew("second", tg);  
System.out.println("Starting the second");  
  
// checking the number of active thread by invoking the activeCount() method  
System.out.println("The total number of active threads are: " + tg.activeCount());  
}  
}
```

Output:

```
Starting the first
Starting the second
The total number of active threads are: 2
```

Thread Pool Methods Example: int activeGroupCount()

Now, we will learn how one can use the activeGroupCount() method in the code.

FileName: ActiveGroupCountExample.java

```
// Java code illustrating the activeGroupCount() method
```

```
// import statement
```

```
import java.lang.*;
```

```
class ThreadNew extends Thread
```

```
{
```

```
// constructor of the class
```

```
ThreadNew(String tName, ThreadGroup tgrp)
```

```
{
```

```
super(tgrp, tName);
```

```
start();
```

```
}
```

```
// overriding the run() method
```

```
public void run()
```

```
{
```

```
for (int j = 0; j < 100; j++)
```

```
{
```

```
try
```

```
{
```

```
Thread.sleep(5);
```

```
}
```

```
catch (InterruptedException e)
```

```
{  
    System.out.println("The exception has been encountered " + e);  
}  
  
}  
  
System.out.println(Thread.currentThread().getName() + " thread has finished executing");  
}  
}  
  
public class ActiveGroupCountExample  
{  
    // main method  
    public static void main(String args[])  
    {  
        // creating the thread group  
        ThreadGroup tg = new ThreadGroup("The parent group of threads");  
  
        ThreadGroup tg1 = new ThreadGroup(tg, "the child group");  
  
        ThreadNew th1 = new ThreadNew("the first", tg);  
        System.out.println("Starting the first");  
  
        ThreadNew th2 = new ThreadNew("the second", tg);  
        System.out.println("Starting the second");  
  
        // checking the number of active thread by invoking the activeGroupCount() method  
        System.out.println("The total number of active thread groups are: " + tg.activeGroupCount());  
    }  
}
```

Output:

AD

```
Starting the first  
Starting the second
```

The total number of active thread groups are: 1
the second thread has finished executing
the first thread has finished executing

Thread Pool Methods Example: void destroy()

Now, we will learn how one can use the destroy() method in the code.

FileName: DestroyExample.java

```
// Code illustrating the destroy() method

// import statement
import java.lang.*;

class ThreadNew extends Thread
{
    // constructor of the class
    ThreadNew(String tName, ThreadGroup tgrp)
    {
        super(tgrp, tName);
        start();
    }

    // overriding the run() method
    public void run()
    {

        for (int j = 0; j < 100; j++)
        {
            try
            {
                Thread.sleep(5);
            }
            catch (InterruptedException e)
            {
                System.out.println("The exception has been encountered " + e);
            }
        }
    }
}
```

```
}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}

}

public class DestroyExample
{
// main method
public static void main(String args[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

// waiting until the other threads has been finished
th1.join();
th2.join();

// destroying the child thread group
tg1.destroy();
System.out.println(tg1.getName() + " is destroyed.");

// destroying the parent thread group
tg.destroy();
System.out.println(tg.getName() + " is destroyed.");
}
```

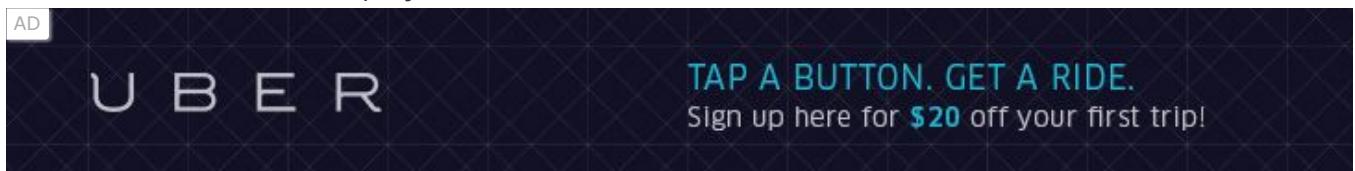
Output:

```
Starting the first
Starting the second
the first thread has finished executing
the second thread has finished executing
the child group is destroyed.
the parent group is destroyed.
```

Thread Pool Methods Example: int enumerate()

Now, we will learn how one can use the enumerate() method in the code.

FileName: EnumerateExample.java



```
// Code illustrating the enumerate() method
```

```
// import statement
```

```
import java.lang.*;
```

```
class ThreadNew extends Thread
```

```
{
```

```
// constructor of the class
```

```
ThreadNew(String tName, ThreadGroup tgrp)
```

```
{
```

```
super(tgrp, tName);
```

```
start();
```

```
}
```

```
// overriding the run() method
```

```
public void run()
```

```
{
```

```
for (int j = 0; j < 100; j++)
```

```
{
```

```
try
```

```
{  
    Thread.sleep(5);  
}  
catch (InterruptedException e)  
{  
    System.out.println("The exception has been encountered " + e);  
}  
  
}  
  
System.out.println(Thread.currentThread().getName() + " thread has finished executing");  
}  
}  
  
public class EnumerateExample  
{  
    // main method  
public static void main(String args[]) throws SecurityException, InterruptedException  
{  
    // creating the thread group  
    ThreadGroup tg = new ThreadGroup("the parent group");  
  
    ThreadGroup tg1 = new ThreadGroup(tg, "the child group");  
  
    ThreadNew th1 = new ThreadNew("the first", tg);  
    System.out.println("Starting the first");  
  
    ThreadNew th2 = new ThreadNew("the second", tg);  
    System.out.println("Starting the second");  
  
    // returning the number of threads kept in this array  
    Thread[] grp = new Thread[tg.activeCount()];  
    int cnt = tg.enumerate(grp);  
    for (int j = 0; j < cnt; j++)  
    {  
        System.out.println("Thread " + grp[j].getName() + " is found.");  
    }  
}
```

Output:

```
Starting the first
Starting the second
Thread the first is found.
Thread the second is found.
the first thread has finished executing
the second thread has finished executing
```

Thread Pool Methods Example: int getMaxPriority()

The following code shows the working of the getMaxPriority() method.

FileName: GetMaxPriorityExample.java

```
// Code illustrating the getMaxPriority() method

// import statement
import java.lang.*;

class ThreadNew extends Thread
{
    // constructor of the class
    ThreadNew(String tName, ThreadGroup tgrp)
    {
        super(tgrp, tName);
        start();
    }

    // overriding the run() method
    public void run()
    {

        for (int j = 0; j < 100; j++)
        {
            try
            {
```

```
Thread.sleep(5);
}

catch (InterruptedException e)
{
    System.out.println("The exception has been encountered " + e);
}

}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}
```

public class GetMaxPriorityExample

```
{
// main method
public static void main(String args[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

int priority = tg.getMaxPriority();

System.out.println("The maximum priority of the parent ThreadGroup: " + priority);

}
```

Output:

```
Starting the first
Starting the second
The maximum priority of the parent ThreadGroup: 10
the first thread has finished executing
the second thread has finished executing
```

Thread Pool Methods Example: ThreadGroup getParent()

Now, we will learn how one can use the getParent() method in the code.

FileName: GetParentExample.java

```
// Code illustrating the getParent() method

// import statement
import java.lang.*;

class ThreadNew extends Thread
{
    // constructor of the class
    ThreadNew(String tName, ThreadGroup tgrp)
    {
        super(tgrp, tName);
        start();
    }

    // overriding the run() method
    public void run()
    {

        for (int j = 0; j < 100; j++)
        {
            try
            {
                Thread.sleep(5);
            }
            catch (InterruptedException e)
        }
    }
}
```

```
{  
    System.out.println("The exception has been encountered" + e);  
}  
  
}  
  
System.out.println(Thread.currentThread().getName() + " thread has finished executing");  
}  
}  
  
public class GetMaxPriorityExample  
{  
    // main method  
    public static void main(String args[]) throws SecurityException, InterruptedException  
    {  
        // creating the thread group  
        ThreadGroup tg = new ThreadGroup("the parent group");  
  
        ThreadGroup tg1 = new ThreadGroup(tg, "the child group");  
  
        ThreadNew th1 = new ThreadNew("the first", tg);  
        System.out.println("Starting the first");  
  
        ThreadNew th2 = new ThreadNew("the second", tg);  
        System.out.println("Starting the second");  
  
        // printing the parent ThreadGroup  
        // of both child and parent threads  
        System.out.println("The ParentThreadGroup for " + tg.getName() + " is " + tg.getParent().getName());  
        System.out.println("The ParentThreadGroup for " + tg1.getName() + " is " + tg1.getParent().getName())  
  
    }  
}
```

Output:

```
Starting the first
Starting the second
The ParentThreadGroup for the parent group is main
The ParentThreadGroup for the child group is the parent group
the first thread has finished executing
the second thread has finished executing
```

Thread Pool Methods Example: void interrupt()

The following program illustrates how one can use the interrupt() method.

FileName: InterruptExample.java

```
// Code illustrating the interrupt() method

// import statement
import java.lang.*;

class ThreadNew extends Thread
{
    // constructor of the class
    ThreadNew(String tName, ThreadGroup tgrp)
    {
        super(tgrp, tName);
        start();
    }

    // overriding the run() method
    public void run()
    {

        for (int j = 0; j < 100; j++)
        {
            try
            {
                Thread.sleep(5);
            }
        }
    }
}
```

```
catch (InterruptedException e)
{
    System.out.println("The exception has been encountered " + e);
}

}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}
```

public class InterruptExample

```
{
// main method
public static void main(String args[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

// invoking the interrupt method
tg.interrupt();

}
}
```

Output:

```
Starting the first
Starting the second
The exception has been encountered java.lang.InterruptedException: sleep interrupted
```

The exception has been encountered java.lang.InterruptedException: sleep interrupted
the second thread has finished executing
the first thread has finished executing

Thread Pool Methods Example: boolean isDaemon()

The following program illustrates how one can use the isDaemon() method.

FileName: IsDaemonExample.java

```
// Code illustrating the isDaemon() method

// import statement
import java.lang.*;

class ThreadNew extends Thread
{
    // constructor of the class
    ThreadNew(String tName, ThreadGroup tgrp)
    {
        super(tgrp, tName);
        start();
    }

    // overriding the run() method
    public void run()
    {

        for (int j = 0; j < 100; j++)
        {
            try
            {
                Thread.sleep(5);
            }
            catch (InterruptedException e)
            {
                System.out.println("The exception has been encountered" + e);
            }
        }
    }
}
```

```
}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}

}

public class IsDaemonExample
{
// main method
public static void main(String args[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

if (tg.isDaemon() == true)
{
System.out.println("The group is a daemon group.");
}

else
{
System.out.println("The group is not a daemon group.");
}

}
```

Output:

```
Starting the first
Starting the second
The group is not a daemon group.
the second thread has finished executing
the first thread has finished executing
```

Thread Pool Methods Example: boolean isDestroyed()

The following program illustrates how one can use the isDestroyed() method.

FileName: IsDestroyedExample.java

```
// Code illustrating the isDestroyed() method
```

```
// import statement
```

```
import java.lang.*;
```

```
class ThreadNew extends Thread
```

```
{
```

```
// constructor of the class
```

```
ThreadNew(String tName, ThreadGroup tgrp)
```

```
{
```

```
super(tgrp, tName);
```

```
start();
```

```
}
```

```
// overriding the run() method
```

```
public void run()
```

```
{
```

```
for (int j = 0; j < 100; j++)
```

```
{
```

```
try
```

```
{
```

```
Thread.sleep(5);
```

```
}
```

```
catch (InterruptedException e)
```

```
{  
    System.out.println("The exception has been encountered" + e);  
}  
  
}  
  
System.out.println(Thread.currentThread().getName() + " thread has finished executing");  
}  
}  
  
public class IsDestroyedExample  
{  
    // main method  
    public static void main(String args[]) throws SecurityException, InterruptedException  
    {  
        // creating the thread group  
        ThreadGroup tg = new ThreadGroup("the parent group");  
  
        ThreadGroup tg1 = new ThreadGroup(tg, "the child group");  
  
        ThreadNew th1 = new ThreadNew("the first", tg);  
        System.out.println("Starting the first");  
  
        ThreadNew th2 = new ThreadNew("the second", tg);  
        System.out.println("Starting the second");  
  
        if (tg.isDestroyed() == true)  
        {  
            System.out.println("The group has been destroyed.");  
        }  
        else  
        {  
            System.out.println("The group has not been destroyed.");  
        }  
    }  
}
```

Output:

```
Starting the first
Starting the second
The group has not been destroyed.
the first thread has finished executing
the second thread has finished executing
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#) [T-SQL tutorial](#)

Transact-SQL

Java Shutdown Hook

A special construct that facilitates the developers to add some code that has to be run when the Java Virtual Machine (JVM) is shutting down is known as the **Java shutdown hook**. The Java shutdown hook comes in very handy in the cases where one needs to perform some special cleanup work when the JVM is shutting down. Note that handling an operation such as invoking a special method before the JVM terminates does not work using a general construct when the JVM is shutting down due to some external factors. For example, whenever a kill request is generated by the operating system or due to resource is not allocated because of the lack of free memory, then in such a case, it is not possible to invoke the procedure. The shutdown hook solves this problem comfortably by providing an arbitrary block of code.

Taking at a surface level, learning about the shutdown hook is straightforward. All one has to do is to derive a class using the `java.lang.Thread` class, and then provide the code for the task one wants to do in the `run()` method when the JVM is going to shut down. For registering the instance of the derived class as the shutdown hook, one has to invoke the method `Runtime.getRuntime().addShutdownHook(Thread)`, whereas for removing the already registered shutdown hook, one has to invoke the `removeShutdownHook(Thread)` method.

In nutshell, the shutdown hook can be used to perform cleanup resources or save the state when JVM shuts down normally or abruptly. Performing clean resources means closing log files, sending some alerts, or something else. So if you want to execute some code before JVM shuts down, use the shutdown hook.

When does the JVM shut down?

The JVM shuts down when:

- user presses `ctrl+c` on the command prompt
- `System.exit(int)` method is invoked
- user logoff
- user shutdown etc.

The `addShutdownHook(Thread hook)` method

The `addShutdownHook()` method of the `Runtime` class is used to register the thread with the Virtual Machine.

Syntax:

```
public void addShutdownHook(Thread hook){}
```

The object of the Runtime class can be obtained by calling the static factory method `getRuntime()`. For example:

```
Runtime r = Runtime.getRuntime();
```

The `removeShutdownHook(Thread hook)` method

The `removeShutdownHook()` method of the Runtime class is invoked to remove the registration of the already registered shutdown hooks.

Syntax:

```
public boolean removeShutdownHook(Thread hook){ }
```

True value is returned by the method, when the method successfully de-register the registered hooks; otherwise returns false.

Factory method

The method that returns the instance of a class is known as factory method.

Simple example of Shutdown Hook

FileName: MyThread.java

```
class MyThread extends Thread{
    public void run(){
        System.out.println("shut down hook task completed..");
    }
}
```

```
public class TestShutdown1{
    public static void main(String[] args) throws Exception {
```

```
        Runtime r=Runtime.getRuntime();
        r.addShutdownHook(new MyThread());
```

```
System.out.println("Now main sleeping... press ctrl+c to exit");
try{Thread.sleep(3000);}catch (Exception e){}
}
}
```

Output:

```
Now main sleeping... press ctrl+c to exit
shut down hook task completed.
```

Same example of Shutdown Hook by anonymous class:

FileName: TestShutdown2.java

```
public class TestShutdown2{
public static void main(String[] args) throws Exception {
    Runtime r=Runtime.getRuntime();
    r.addShutdownHook(new Thread(){
        public void run(){
            System.out.println("shut down hook task completed..");
        }
    });
}

System.out.println("Now main sleeping... press ctrl+c to exit");
try{Thread.sleep(3000);}catch (Exception e){}
}
}
```

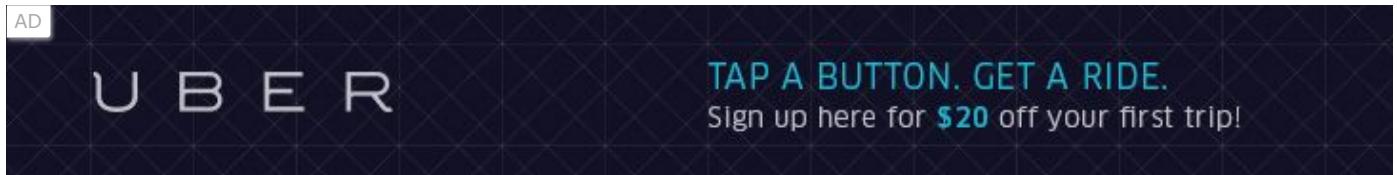
Output:

```
Now main sleeping... press ctrl+c to exit
shut down hook task completed.
```

Removing the registered shutdown hook example

The following example shows how one can use the removeShutdownHook() method to remove the registered shutdown hook.

FileName: RemoveHookExample.java



```
public class RemoveHookExample
{
    // the Msg class is derived from the Thread class
    static class Msg extends Thread
    {
        public void run()
        {
            System.out.println("Bye ...");
        }
    }

    // main method
    public static void main(String[] args)
    {
        try
        {
            // creating an object of the class Msg
            Msg ms = new Msg();

            // registering the Msg object as the shutdown hook
            Runtime.getRuntime().addShutdownHook(ms);

            // printing the current state of program
            System.out.println("The program is beginning ...");

            // causing the thread to sleep for 2 seconds
        }
    }
}
```

```
System.out.println("Waiting for 2 seconds ...");
Thread.sleep(2000);

// removing the hook
Runtime.getRuntime().removeShutdownHook(ms);

// printing the message program is terminating
System.out.println("The program is terminating ...");
}

catch (Exception ex)
{
ex.printStackTrace();
}
}
```

Output:

```
The program is beginning ...
Waiting for 2 seconds ...
The program is terminating ...
```

Points to Remember

There are some important points to keep in mind while working with the shutdown hook.

No guarantee for the execution of the shutdown hooks: The first and the most important thing to keep in mind is that there is no certainty about the execution of the shutdown hook. In some scenarios, the shutdown hooks will not execute at all. For example, if the JVM gets crashed due to some internal error, then there is no scope for the shutdown hooks. When the operating system gives the SYSKILL signal, then also it is not possible for the shutdown hooks to come into picture.

Note that when the application is terminated normally the shutdown hooks are called (all threads of the application is finished or terminated). Also, when the operating system is shut down or the user presses the **ctrl + c** the shutdown hooks are invoked.

Before completion, the shutdown hooks can be stopped forcefully: It is a special case of the above discussed point. Whenever a shutdown hooks start to execute, one can forcefully terminate it by shutting down the system. In this case, the operating system for a specific amount of time. If the job

is not done in that frame of time, then the system has no other choice than to forcefully terminate the running hooks.

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

There can be more than one shutdown hooks, but their execution order is not guaranteed: The JVM can execute the shutdown hooks in any arbitrary order. Even concurrent execution of the shutdown hooks are also possible.

Within shutdown hooks, it is not allowed to unregister or register the shutdown hooks: When the JVM initiates the shutdown sequence, one can not remove or add more any existing shutdown hooks. If one tries to do so, the `IllegalStateException` is thrown by the JVM.

The `Runtime.halt()` can stop the shutdown sequence that has been started: Only the `Runtime.halt()`, which terminates the JVM forcefully, can stop the started shutdown sequence, which also means that invoking the `System.exit()` method will not work within a shutdown hook.

Security permissions are required when using shutdown hooks: If one is using the Java Security Managers, then the Java code that is responsible for removing or adding the shutdown hooks need to get the shutdown hooks permission at the runtime. If one invokes the method without getting the permission in the secure environment, then it will raise the `SecurityException`.

← Prev

Next →

AD

[For Videos Join Our YouTube Channel: Join Now](#)

How to perform single task by multiple threads in Java?

If you have to perform a single task by many threads, have only one run() method. For example:

Program of performing single task by multiple threads

FileName: TestMultitasking1.java

```
class TestMultitasking1 extends Thread{  
    public void run(){  
        System.out.println("task one");  
    }  
    public static void main(String args[]){  
        TestMultitasking1 t1=new TestMultitasking1();  
        TestMultitasking1 t2=new TestMultitasking1();  
        TestMultitasking1 t3=new TestMultitasking1();  
  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

Test it Now

Output:

```
task one  
task one  
task one
```

Program of performing single task by multiple threads

FileName: TestMultitasking2.java

```
class TestMultitasking2 implements Runnable{
```

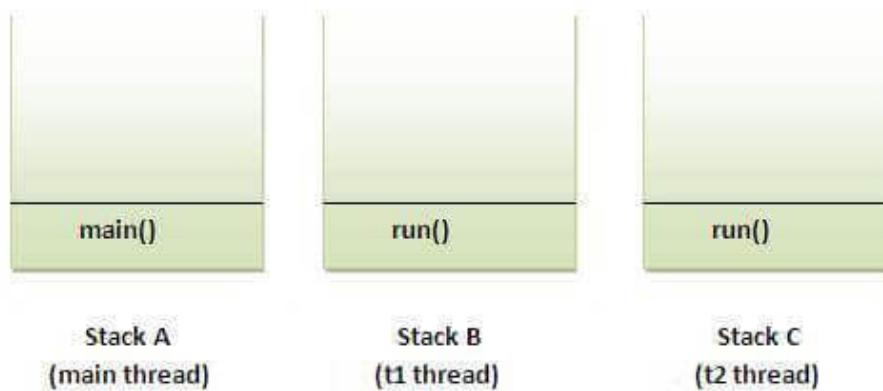
```
public void run(){  
    System.out.println("task one");  
}  
  
public static void main(String args[]){  
    Thread t1 = new Thread(new TestMultitasking2()); //passing anonymous object of TestMultitasking  
    Thread t2 = new Thread(new TestMultitasking2());  
  
    t1.start();  
    t2.start();  
  
}  
}
```

Test it Now

Output:

```
task one  
task one
```

Note: Each thread runs in a separate callstack.



How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads, have multiple run() methods. For example:

Program of performing two tasks by two threads

FileName: TestMultitasking3.java

```
class Simple1 extends Thread{
    public void run(){
        System.out.println("task one");
    }
}

class Simple2 extends Thread{
    public void run(){
        System.out.println("task two");
    }
}

class TestMultitasking3{
    public static void main(String args[]){
        Simple1 t1=new Simple1();
        Simple2 t2=new Simple2();

        t1.start();
        t2.start();
    }
}
```

Test it Now

Output:

```
task one
task two
```

Same example as above by anonymous class that extends Thread class:

Program of performing two tasks by two threads

FileName: TestMultitasking4.java

```
class TestMultitasking4{
    public static void main(String args[]){
        Thread t1=new Thread(){
            public void run(){
                System.out.println("task one");
            }
        };
        Thread t2=new Thread(){
            public void run(){
                System.out.println("task two");
            }
        };

        t1.start();
        t2.start();
    }
}
```

Test it Now

Output:

```
task one
task two
```

Same example as above by anonymous class that implements Runnable interface:

Program of performing two tasks by two threads

FileName: TestMultitasking5.java

```
class TestMultitasking5{
```

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

```
    Runnable r1=new Runnable(){
```

```
        public void run(){
```

```
            System.out.println("task one");
```

```
}
```

```
};
```

```
Runnable r2=new Runnable(){
```

```
        public void run(){
```

```
            System.out.println("task two");
```

```
}
```

```
};
```

```
Thread t1=new Thread(r1);
```

```
Thread t2=new Thread(r2);
```

```
    t1.start();
```

```
    t2.start();
```

```
}
```

```
}
```

Test it Now

Output:

```
task one
```

```
task two
```

Printing even and odd numbers using two threads

To print the even and odd numbers using the two threads, we will use the synchronized block and the notify() method. Observe the following program.

FileName: OddEvenExample.java

AD

```
// Java program that prints the odd and even numbers using two threads.  
// the time complexity of the program is O(N), where N is the number up to which we  
// are displaying the numbers  
public class OddEvenExample  
{  
    // Starting the counter  
    int contr = 1;  
    static int NUM;  
    // Method for printing the odd numbers  
    public void displayOddNumber()  
    {  
        // note that synchronized blocks are necessary for the code for getting the desired  
        // output. If we remove the synchronized blocks, we will get an exception.  
        synchronized (this)  
        {  
            // Printing the numbers till NUM  
            while (contr < NUM)  
            {  
                // If the contr is even then display  
                while (contr % 2 == 0)  
                {  
                    // handling the exception handle  
                    try  
                    {  
                        wait();  
                    }  
                    catch (InterruptedException ex)  
                    {  
                        ex.printStackTrace();  
                    }  
                }  
                // Printing the number  
                System.out.print(contr + " ");  
                // Incrementing the contr  
                contr = contr + 1;  
                // notifying the thread which is waiting for this lock  
                notify();  
            }  
        }  
    }  
}
```

```
}

}

}

// Method for printing the even numbers

public void displayEvenNumber()

{

synchronized (this)

{

// Printing the number till NUM

while (contr < NUM)

{

// If the count is odd then display

while (contr % 2 == 1)

{

// handling the exception

try

{

wait();

}

catch (InterruptedException ex)

{

ex.printStackTrace();

}

}

// Printing the number

System.out.print(contr + " ");

// Incrementing the contr

contr = contr +1;

// Notifying to the 2nd thread

notify();

}

}

}

// main method

public static void main(String[] args)

{

// The NUM is given

NUM = 20;
```

```
// creating an object of the class OddEvenExample
OddEvenExample oe = new OddEvenExample();

// creating a thread th1
Thread th1 = new Thread(new Runnable()

{
    public void run()
    {
        // invoking the method displayEvenNumber() using the thread th1
        oe.displayEvenNumber();
    }
});

// creating a thread th2
Thread th2 = new Thread(new Runnable()

{
    public void run()
    {
        // invoking the method displayOddNumber() using the thread th2
        oe.displayOddNumber();
    }
});

// starting both of the threads
th1.start();
th2.start();
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

← Prev

Next →

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

AD

How can an object be unreferenced?

01

By nulling the reference

02

By assigning a reference to another

03

By anonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}
```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{  
    public void finalize(){System.out.println("object is garbage collected");}  
    public static void main(String args[]){  
        TestGarbage1 s1=new TestGarbage1();  
        TestGarbage1 s2=new TestGarbage1();  
        s1=null;  
        s2=null;  
        System.gc();  
    }  
}
```

Test it Now

```
object is garbage collected  
object is garbage collected
```

Note: Neither finalization nor garbage collection is guaranteed.

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

Java Runtime class

Java Runtime class is used to interact with java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

Important methods of Java Runtime class

No.	Method	Description
1)	public static Runtime getRuntime()	returns the instance of Runtime class.
2)	public void exit(int status)	terminates the current virtual machine.
3)	public void addShutdownHook(Thread hook)	registers new hook thread.
4)	public Process exec(String command)throws IOException	executes given command in a separate process.
5)	public int availableProcessors()	returns no. of available processors.
6)	public long freeMemory()	returns amount of free memory in JVM.
7)	public long totalMemory()	returns amount of total memory in JVM.

Java Runtime exec() method

```
public class Runtime1{
    public static void main(String args[])throws Exception{
        Runtime.getRuntime().exec("notepad");//will open a new notepad
    }
}
```

How to shutdown system in Java

You can use `shutdown -s` command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. `c:\\Windows\\System32\\shutdown`.

Here you can use `-s` switch to shutdown system, `-r` switch to restart system and `-t` switch to specify time delay.

```
public class Runtime2{  
    public static void main(String args[])throws Exception{  
        Runtime.getRuntime().exec("shutdown -s -t 0");  
    }  
}
```



How to shutdown windows system in Java

```
public class Runtime2{  
    public static void main(String args[])throws Exception{  
        Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");  
    }  
}
```

How to restart system in Java

```
public class Runtime3{  
    public static void main(String args[])throws Exception{  
        Runtime.getRuntime().exec("shutdown -r -t 0");  
    }  
}
```

Java Runtime availableProcessors()

```
public class Runtime4{  
    public static void main(String args[])throws Exception{  
        System.out.println(Runtime.getRuntime().availableProcessors());  
    }  
}
```

Java Runtime freeMemory() and totalMemory() method

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

```
public class MemoryTest{  
    public static void main(String args[])throws Exception{  
        Runtime r=Runtime.getRuntime();  
        System.out.println("Total Memory: "+r.totalMemory());  
        System.out.println("Free Memory: "+r.freeMemory());  
  
        for(int i=0;i<10000;i++){  
            new MemoryTest();  
        }  
        System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());  
        System.gc();  
        System.out.println("After gc(), Free Memory: "+r.freeMemory());  
    }  
}
```

```
Total Memory: 100139008  
Free Memory: 99474824  
After creating 10000 instance, Free Memory: 99310552  
After gc(), Free Memory: 100182832
```

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
 1. Synchronized method.
 2. Synchronized block.
 3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package `java.util.concurrent.locks` contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

TestSynchronization1.java

```
class Table{  
    void printTable(int n){//method not synchronized  
        for(int i=1;i<=5;i++){  
            System.out.println(n*i);  
            try{  
                Thread.sleep(400);  
            }catch(Exception e){System.out.println(e);}  
        }  
    }  
}
```

```
class MyThread1 extends Thread{  
    Table t;  
    MyThread1(Table t){  
        this.t=t;  
    }  
    public void run(){  
        t.printTable(5);  
    }  
}
```

```
}

}

class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
    this.t=t;
}
public void run(){
    t.printTable(100);
}
}

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

Java Synchronized Method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TestSynchronization2.java

```
//example of java synchronized method
class Table{
    synchronized void printTable(int n){//synchronized method
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }
    }

    class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
            this.t=t;
        }
        public void run(){
            t.printTable(5);
        }
    }

    class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
            this.t=t;
        }
    }
}
```

```
public void run(){
    t.printTable(100);
}

public class TestSynchronization2{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

Output:



```
5
10
15
20
25
100
200
300
400
500
```

Example of synchronized method by using anonymous class

In this program, we have created the two threads by using the anonymous class, so less coding is required.

TestSynchronization3.java

```
//Program of synchronized method by using anonymous class

class Table{
    synchronized void printTable(int n){//synchronized method
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }
    }

    public class TestSynchronization3{
        public static void main(String args[]){
            final Table obj = new Table();//only one object

            Thread t1=new Thread(){
                public void run(){
                    obj.printTable(5);
                }
            };

            Thread t2=new Thread(){
                public void run(){
                    obj.printTable(100);
                }
            };

            t1.start();
            t2.start();
        }
    }
}
```

Output:

5
10
15
20
25
100
200
300
400
500

Next →

AD

 [Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keyword.
- Java synchronized block is more efficient than Java synchronized method.

Syntax

```
synchronized (object reference expression) {  
    //code block  
}
```

Example of Synchronized Block

Let's see the simple example of synchronized block.

TestSynchronizedBlock1.java

```
class Table  
{  
    void printTable(int n){  
        synchronized(this)//synchronized block  
        {  
            for(int i=1;i<=5;i++){  
                System.out.println(n*i);  
            }  
        }  
    }  
}
```

```
try{
    Thread.sleep(400);
}catch(Exception e){System.out.println(e);}
}
}
}//end of the method
}
```

```
class MyThread1 extends Thread{
```

```
Table t;
MyThread1(Table t){
    this.t=t;
}
public void run(){
    t.printTable(5);
}
```

```
}
```

```
class MyThread2 extends Thread{
```

```
Table t;
MyThread2(Table t){
    this.t=t;
}
public void run(){
    t.printTable(100);
}
```

```
public class TestSynchronizedBlock1{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

Output:

```
5  
10  
15  
20  
25  
100  
200  
300  
400  
500
```

Synchronized Block Example Using Anonymous Class

TestSynchronizedBlock2.java

```
// A Sender class  
class Sender  
{  
    public void SenderMsg(String msg)  
    {  
        System.out.println("\nSending a Message: " + msg);  
        try  
        {  
            Thread.sleep(800);  
        }  
        catch (Exception e)  
        {  
            System.out.println("Thread interrupted.");  
        }  
        System.out.println("\n" + msg + "Sent");  
    }  
}  
  
// A Sender class for sending a message using Threads  
class SenderWThreads extends Thread  
{  
    private String msg;
```

```
Sender sd;

// Receiver method to receive a message object and a message to be sent
SenderWThreads(String m, Sender obj)
{
    msg = m;
    sd = obj;
}

public void run()
{
    // Checks that only one thread sends a message at a time.
    synchronized(sd)
    {
        // synchronizing the sender object
        sd.SenderMsg(msg);
    }
}
}

// Driver Code
public class ShynchronizedMultithreading
{
    public static void main(String args[])
    {
        Sender sender = new Sender();
        SenderWThreads sender1 = new SenderWThreads( "Hola " , sender);
        SenderWThreads sender2 = new SenderWThreads( "Welcome to Javatpoint website " , sender);

        // Start two threads of SenderWThreads type
        sender1.start();
        sender2.start();

        // wait for threads to end
        try
        {
            sender1.join();
            sender2.join();
        }
    }
}
```

```
catch(Exception e)
{
    System.out.println("Interrupted");
}
}
```

Output:

```
Sending a Message: Hola
Hola Sent
Sending a Message: Welcome to Javatpoint website
Welcome to Javatpoint website Sent
```

← Prev

Next →

AD

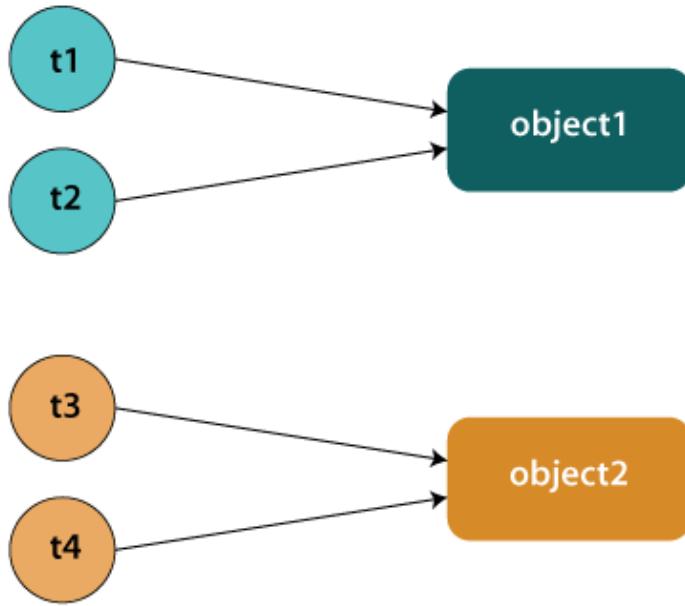
 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named **object1** and **object2**. In case of synchronized method and synchronized block there cannot be interference between **t1** and **t2** or **t3** and **t4** because **t1** and **t2** both refers to a common object that have a single lock. But there can be interference between **t1** and **t3** or **t2** and **t4** because **t1** acquires another lock and **t3** acquires another lock. We don't want interference between **t1** and **t3** or **t2** and **t4**. Static synchronization solves this problem.

Example of Static Synchronization

In this example we have used **synchronized** keyword on the static method to perform static synchronization.

TestSynchronization4.java

```
class Table
{
    synchronized static void printTable(int n){
        for(int i=1;i<=10;i++){
            System.out.println(n*i);
        }
    }
}
```

```
        Thread.sleep(400);

    }catch(Exception e){

    }

}

class MyThread1 extends Thread{

public void run(){

Table.printTable(1);

}

}

class MyThread2 extends Thread{

public void run(){

Table.printTable(10);

}

}

class MyThread3 extends Thread{

public void run(){

Table.printTable(100);

}

}

class MyThread4 extends Thread{

public void run(){

Table.printTable(1000);

}

}

public class TestSynchronization4{

public static void main(String t[]){

MyThread1 t1=new MyThread1();

MyThread2 t2=new MyThread2();

MyThread3 t3=new MyThread3();

MyThread4 t4=new MyThread4();

t1.start();

t2.start();

t3.start();

t4.start();

}

}
```

Test it Now**Output:**

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
10  
20  
30  
40  
50  
60  
70  
80  
90  
100  
100  
200  
300  
400  
500  
600  
700  
800  
900  
1000  
1000  
2000  
3000  
4000  
5000  
6000
```

```
7000  
8000  
9000  
10000
```

Example of static synchronization by Using the anonymous class

In this example, we are using anonymous class to create the threads.

TestSynchronization5.java

```
class Table{  
  
    synchronized static void printTable(int n){  
        for(int i=1;i<=10;i++){  
            System.out.println(n*i);  
            try{  
                Thread.sleep(400);  
            }catch(Exception e){  
            }  
        }  
    }  
}
```

```
public class TestSynchronization5 {  
    public static void main(String[] args) {
```

```
        Thread t1=new Thread(){  
            public void run(){  
                Table.printTable(1);  
            }  
        };
```

```
        Thread t2=new Thread(){  
            public void run(){  
                Table.printTable(10);  
            }  
        };
```

```
Thread t3=new Thread(){
    public void run(){
        Table.printTable(100);
    }
};
```

```
Thread t4=new Thread(){
    public void run(){
        Table.printTable(1000);
    }
};

t1.start();
t2.start();
t3.start();
t4.start();

}
}
```

Test it Now

Output:

```
1
2
3
4
5
6
7
8
9
10
10
20
30
40
```

```
50  
60  
70  
80  
90  
100  
100  
200  
300  
400  
500  
600  
700  
800  
900  
1000  
1000  
2000  
3000  
4000  
5000  
6000  
7000  
8000  
9000  
10000
```

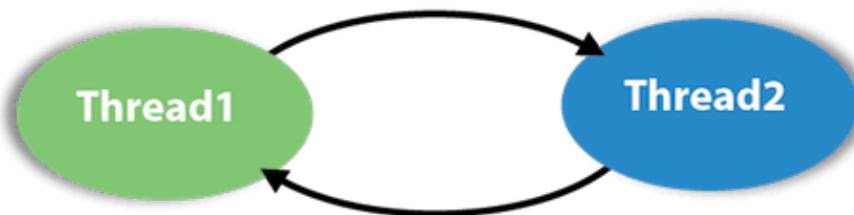
Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

```
static void printTable(int n) {  
    synchronized (Table.class) { // Synchronized block on class A  
        // ...  
    }  
}
```

Deadlock in Java

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in Java

TestDeadlockExample1.java

```
public class TestDeadlockExample1 {  
    public static void main(String[] args) {  
        final String resource1 = "ratan jaiswal";  
        final String resource2 = "vimal jaiswal";  
        // t1 tries to lock resource1 then resource2  
        Thread t1 = new Thread() {  
            public void run() {  
                synchronized (resource1) {  
                    System.out.println("Thread 1: locked resource 1");  
  
                    try { Thread.sleep(100); } catch (Exception e) {}  
  
                    synchronized (resource2) {  
                        System.out.println("Thread 1: locked resource 2");  
                    }  
                }  
            }  
        };  
  
        // t2 tries to lock resource2 then resource1  
    }  
}
```

```
Thread t2 = new Thread() {
```

```
    public void run() {
```

```
        synchronized (resource2) {
```

```
            System.out.println("Thread 2: locked resource 2");
```

```
            try { Thread.sleep(100);} catch (Exception e) {}
```

```
        synchronized (resource1) {
```

```
            System.out.println("Thread 2: locked resource 1");
```

```
        }
```

```
}
```

```
};
```

```
t1.start();
```

```
t2.start();
```

```
}
```

```
}
```

Output:

```
Thread 1: locked resource 1
```

```
    Thread 2: locked resource 2
```

More Complicated Deadlocks

A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

How to avoid deadlock?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

DeadlockSolved.java

```
public class DeadlockSolved {  
  
    public static void main(String ar[]) {  
        DeadlockSolved test = new DeadlockSolved();  
  
        final resource1 a = test.new resource1();  
        final resource2 b = test.new resource2();  
  
        // Thread-1  
        Runnable b1 = new Runnable() {  
            public void run() {  
                synchronized (b) {  
                    try {  
                        /* Adding delay so that both threads can start trying to lock resources */  
                        Thread.sleep(100);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                    // Thread-1 have resource1 but need resource2 also  
                    synchronized (a) {  
                        System.out.println("In block 1");  
                    }  
                }  
            }  
        };  
  
        // Thread-2  
        Runnable b2 = new Runnable() {  
            public void run() {  
                synchronized (b) {  
  
    
```

```
// Thread-2 have resource2 but need resource1 also  
synchronized (a) {  
    System.out.println("In block 2");  
}  
}  
}  
};
```

```
new Thread(b1).start();  
new Thread(b2).start();  
}
```

```
// resource1  
private class resource1 {  
    private int i = 10;
```

```
    public int getI() {  
        return i;  
    }
```

```
    public void setI(int i) {  
        this.i = i;  
    }  
}
```

```
// resource2  
private class resource2 {  
    private int i = 20;
```

```
    public int getI() {  
        return i;  
    }
```

```
    public void setI(int i) {  
        this.i = i;  
    }  
}
```

{

Output:

```
In block 1  
In block 2
```

In the above code, class DeadlockSolved solves the deadlock kind of situation. It will help in avoiding deadlocks, and if encountered, in resolving them.

How to Avoid Deadlock in Java?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks:** We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks:** The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join:** A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use **join** with a maximum time that a thread will take.

[← Prev](#)[Next →](#)

AD



Inter-thread Communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait() throws InterruptedException	It waits until object is notified.
public final void wait(long timeout) throws InterruptedException	It waits for the specified amount of time.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

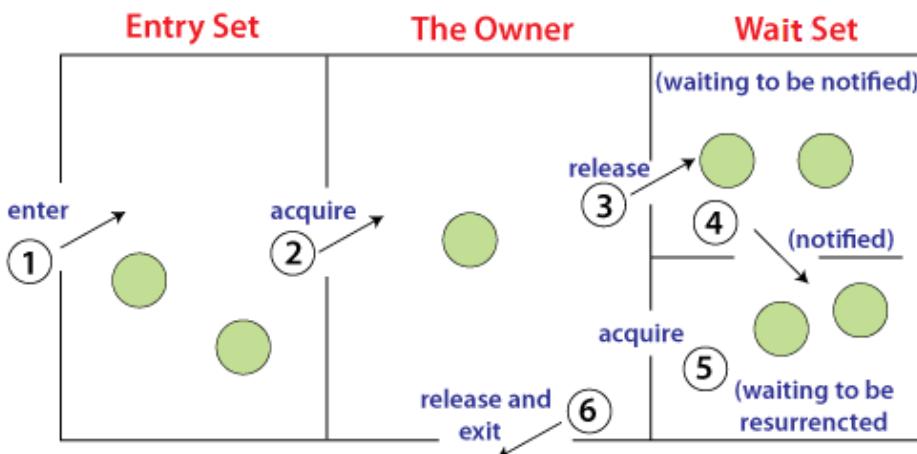
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call `wait()` method on the object. Otherwise it releases the lock and exits.
4. If you call `notify()` or `notifyAll()` method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why `wait()`, `notify()` and `notifyAll()` methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
class Customer{
    int amount=10000;

    synchronized void withdraw(int amount){
        System.out.println("going to withdraw...");

        if(this.amount<amount){
            System.out.println("Less balance; waiting for deposit...");
            try{wait();}catch(Exception e){}
        }

        this.amount-=amount;
        System.out.println("withdraw completed...");
    }

    synchronized void deposit(int amount){
        System.out.println("going to deposit...");
        this.amount+=amount;
        System.out.println("deposit completed... ");
        notify();
    }
}
```

```
class Test{
    public static void main(String args[]){
        final Customer c=new Customer();
        new Thread(){
            public void run(){c.withdraw(15000);}
        }.start();
        new Thread(){
            public void run(){c.deposit(10000);}
        }.start();
    }
}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

TestInterruptingThread1.java

```
class TestInterruptingThread1 extends Thread{
    public void run(){
        try{
            Thread.sleep(1000);
            System.out.println("task");
        }catch(InterruptedException e){
            throw new RuntimeException("Thread interrupted..."+e);
        }
    }

    public static void main(String args[]){
        TestInterruptingThread1 t1=new TestInterruptingThread1();
        t1.start();
        try{
            t1.interrupt();
        }
    }
}
```

```
 }catch(Exception e){System.out.println("Exception handled "+e);}  
}  
}
```

Test it Now

[download this example](#)

Output:

```
Exception in thread-0  
java.lang.RuntimeException: Thread interrupted...  
java.lang.InterruptedException: sleep interrupted  
at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

TestInterruptingThread2.java

```
class TestInterruptingThread2 extends Thread{  
public void run(){  
try{  
Thread.sleep(1000);  
System.out.println("task");  
}catch(InterruptedException e){  
System.out.println("Exception handled "+e);  
}  
System.out.println("thread is running...");  
}  
  
public static void main(String args[]){  
TestInterruptingThread2 t1=new TestInterruptingThread2();  
t1.start();  
  
t1.interrupt();
```

```
}
```

Test it Now

[download this example](#)

Output:

```
Exception handled
    java.lang.InterruptedException: sleep interrupted
        thread is running...
```

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the `interrupt()` method sets the `interrupted` flag to true that can be used to stop the thread by the java programmer later.

TestInterruptingThread3.java

```
class TestInterruptingThread3 extends Thread{

    public void run(){
        for(int i=1;i<=5;i++)
            System.out.println(i);
    }

    public static void main(String args[]){
        TestInterruptingThread3 t1=new TestInterruptingThread3();
        t1.start();

        t1.interrupt();

    }
}
```

Test it Now

Output:

```
1  
2  
3  
4  
5
```



What about isInterrupted and interrupted method?

The `isInterrupted()` method returns the interrupted flag either true or false. The static `interrupted()` method returns the interrupted flag after that it sets the flag to false if it is true.

TestInterruptingThread4.java

```
public class TestInterruptingThread4 extends Thread{  
  
    public void run(){  
        for(int i=1;i<=2;i++){  
            if(Thread.interrupted()){  
                System.out.println("code for interrupted thread");  
            }  
            else{  
                System.out.println("code for normal thread");  
            }  
  
        }//end of for loop  
    }  
  
    public static void main(String args[]){  
  
        TestInterruptingThread4 t1=new TestInterruptingThread4();  
        TestInterruptingThread4 t2=new TestInterruptingThread4();  
  
        t1.start();  
        t1.interrupt();
```

```
t2.start();  
  
}  
}
```

Test it Now

Output:

```
Code for interrupted thread  
code for normal thread  
code for normal thread  
code for normal thread
```

[download this example](#)

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Reentrant Monitor in Java

According to Sun Microsystems, **Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

```
class Reentrant {  
    public synchronized void m() {  
        n();  
        System.out.println("this is m() method");  
    }  
    public synchronized void n() {  
        System.out.println("this is n() method");  
    }  
}
```

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Now let's call the m() method on a thread. In the class given below, we are creating thread using anonymous class.

```
public class ReentrantExample{  
    public static void main(String args[]){  
        final ReentrantExample re=new ReentrantExample();  
  
        Thread t1=new Thread(){  
            public void run(){  
                re.m();//calling method of Reentrant class  
            }  
        };  
        t1.start();  
    }  
}
```

}}

Test it Now

```
Output: this is n() method  
this is m() method
```

<<Prev

Next>>

AD

 [For Videos Join Our Youtube Channel: Join Now](#)**Feedback**

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share**Learn Latest Tutorials** [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

Java SequenceInputStream Class

Java SequenceInputStream class is used to read data from multiple streams. It reads data sequentially (one by one).

Java SequenceInputStream Class declaration

Let's see the declaration for Java.io.SequenceInputStream class:

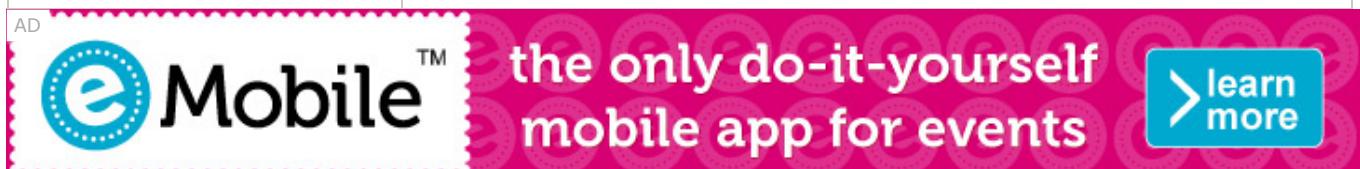
```
public class SequenceInputStream extends InputStream
```

Constructors of SequenceInputStream class

Constructor	Description
SequenceInputStream(InputStream s1, InputStream s2)	creates a new input stream by reading the data of two input stream in order, first s1 and then s2.
SequenceInputStream(Enumeration e)	creates a new input stream by reading the data of an enumeration whose type is InputStream.

Methods of SequenceInputStream class

Method	Description
int read()	It is used to read the next byte of data from the input stream.
int read(byte[] ary, int off, int len)	It is used to read len bytes of data from the input stream into the array of bytes.
int available()	It is used to return the maximum number of byte that can be read from an input stream.
void close()	It is used to close the input stream.



Java SequenceInputStream Example

In this example, we are printing the data of two files testin.txt and testout.txt.

```
package com.javatpoint;

import java.io.*;
class InputStreamExample {
    public static void main(String args[])throws Exception{
        FileInputStream input1=new FileInputStream("D:\\testin.txt");
        FileInputStream input2=new FileInputStream("D:\\testout.txt");
        SequenceInputStream inst=new SequenceInputStream(input1, input2);
        int j;
        while((j=inst.read())!=-1){
            System.out.print((char)j);
        }
        inst.close();
        input1.close();
        input2.close();
    }
}
```

Here, we are assuming that you have two files: testin.txt and testout.txt which have following information:

testin.txt:

```
Welcome to Java IO Programming.
```

testout.txt:

```
It is the example of Java SequenceInputStream class.
```

After executing the program, you will get following output:

Output:

```
Welcome to Java IO Programming. It is the example of Java SequenceInputStream class.
```

Example that reads the data from two files and writes into another file

In this example, we are writing the data of two files **testin1.txt** and **testin2.txt** into another file named **testout.txt**.

```
package com.javatpoint;

import java.io.*;
class Input1{
    public static void main(String args[])throws Exception{
        FileInputStream fin1=new FileInputStream("D:\\testin1.txt");
        FileInputStream fin2=new FileInputStream("D:\\testin2.txt");
        FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
        SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
        int i;
        while((i=sis.read())!=-1)
        {
            fout.write(i);
        }
        sis.close();
        fout.close();
        fin1.close();
        fin2.close();
        System.out.println("Success..");
    }
}
```

Output:

```
Success...
```

testout.txt:

```
Welcome to Java IO Programming. It is the example of Java SequenceInputStream class.
```

SequenceInputStream example that reads data using enumeration

If we need to read the data from more than two files, we need to use [Enumeration](#). Enumeration object can be obtained by calling elements() method of the Vector class. Let's see the simple example where we are reading the data from 4 files: a.txt, b.txt, c.txt and d.txt.

```
package com.javatpoint;
import java.io.*;
import java.util.*;
class Input2{
    public static void main(String args[]) throws IOException{
        //creating the FileInputStream objects for all the files
        FileInputStream fin=new FileInputStream("D:\\a.txt");
        FileInputStream fin2=new FileInputStream("D:\\b.txt");
        FileInputStream fin3=new FileInputStream("D:\\c.txt");
        FileInputStream fin4=new FileInputStream("D:\\d.txt");
        //creating Vector object to all the stream
        Vector v=new Vector();
        v.add(fin);
        v.add(fin2);
        v.add(fin3);
        v.add(fin4);
        //creating enumeration object by calling the elements method
        Enumeration e=v.elements();
        //passing the enumeration object in the constructor
        SequenceInputStream bin=new SequenceInputStream(e);
        int i=0;
        while((i=bin.read())!=-1){
            System.out.print((char)i);
        }
        bin.close();
        fin.close();
        fin2.close();
    }
}
```

The a.txt, b.txt, c.txt and d.txt have following information:

a.txt:

Welcome

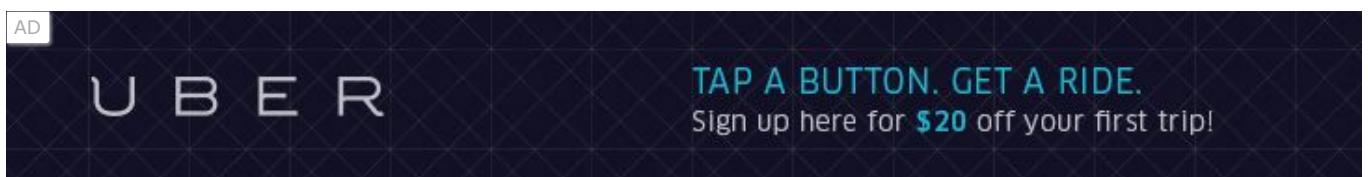
b.txt:

to

c.txt:

java

d.txt:



programming

Output:

Welcome to java programming

← Prev

Next →

AD

Java ByteArrayOutputStream Class

Java ByteArrayOutputStream class is used to **write common data** into multiple files. In this stream, the data is written into a byte **array** which can be written to multiple streams later.

The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

Java ByteArrayOutputStream class declaration

Let's see the declaration for Java.io.ByteArrayOutputStream class:

```
public class ByteArrayOutputStream extends OutputStream
```

Java ByteArrayOutputStream class constructors

Constructor	Description
ByteArrayOutputStream()	Creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int size)	Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Java ByteArrayOutputStream class methods

Method	Description
int size()	It is used to returns the current size of a buffer.
byte[] toByteArray()	It is used to create a newly allocated byte array.
String toString()	It is used for converting the content into a string decoding bytes using a platform default character set.
String toString(String charsetName)	It is used for converting the content into a string decoding bytes using a specified charsetName.

void write(int b)	It is used for writing the byte specified to the byte array output stream.
void write(byte[] b, int off, int len)	It is used for writing len bytes from specified byte array starting from the offset off to the byte array output stream.
void writeTo(OutputStream out)	It is used for writing the complete content of a byte array output stream to the specified output stream.
void reset()	It is used to reset the count field of a byte array output stream to zero value.
void close()	It is used to close the ByteArrayOutputStream.



Example of Java ByteArrayOutputStream

Let's see a simple example of [java](#) ByteArrayOutputStream class to write common data into 2 files: f1.txt and f2.txt.

```
package com.javatpoint;
import java.io.*;
public class DataStreamExample {
    public static void main(String args[]) throws Exception{
        FileOutputStream fout1=new FileOutputStream("D:\\f1.txt");
        FileOutputStream fout2=new FileOutputStream("D:\\f2.txt");

        ByteArrayOutputStream bout=new ByteArrayOutputStream();
        bout.write(65);
        bout.writeTo(fout1);
        bout.writeTo(fout2);

        bout.flush();
        bout.close(); //has no effect
        System.out.println("Success...");
    }
}
```

```
}
```

Output:

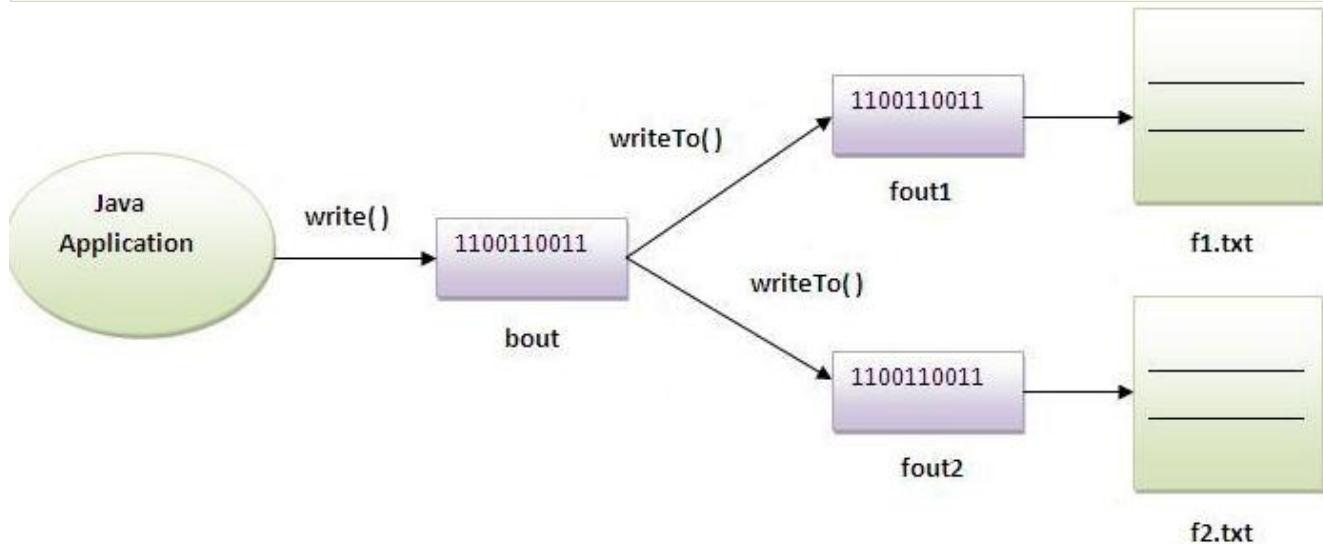
```
Success...
```

f1.txt:

```
A
```

f2.txt:

```
A
```



← Prev

Next →

AD

Java ByteArrayInputStream Class

The `ByteArrayInputStream` is composed of two words: `ByteArray` and `InputStream`. As the name suggests, it can be used to read byte `array` as input stream.

Java `ByteArrayInputStream` `class` contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

The buffer of `ByteArrayInputStream` automatically grows according to data.

Java `ByteArrayInputStream` class declaration

Let's see the declaration for `Java.io.ByteArrayInputStream` class:

```
public class ByteArrayInputStream extends InputStream
```

Java `ByteArrayInputStream` class constructors

Constructor	Description
<code>ByteArrayInputStream(byte[] ary)</code>	Creates a new byte array input stream which uses <code>ary</code> as its buffer array.
<code>ByteArrayInputStream(byte[] ary, int offset, int len)</code>	Creates a new byte array input stream which uses <code>ary</code> as its buffer array that can read up to specified <code>len</code> bytes of data from an array.

Java `ByteArrayInputStream` class methods

Methods	Description
<code>int available()</code>	It is used to return the number of remaining bytes that can be read from the input stream.
<code>int read()</code>	It is used to read the next byte of data from the input stream.
<code>int read(byte[] ary, int off, int len)</code>	It is used to read up to <code>len</code> bytes of data from an array of bytes in the input stream.

boolean markSupported()	It is used to test the input stream for mark and reset method.
long skip(long x)	It is used to skip the x bytes of input from the input stream.
void mark(int readAheadLimit)	It is used to set the current marked position in the stream.
void reset()	It is used to reset the buffer of a byte array.
void close()	It is used for closing a ByteArrayInputStream.



Example of Java ByteArrayInputStream

Let's see a simple example of [java](#) `ByteArrayInputStream` class to read byte array as input stream.

```
package com.javatpoint;
import java.io.*;
public class ReadExample {
    public static void main(String[] args) throws IOException {
        byte[] buf = { 35, 36, 37, 38 };
        // Create the new byte array input stream
        ByteArrayInputStream byt = new ByteArrayInputStream(buf);
        int k = 0;
        while ((k = byt.read()) != -1) {
            //Conversion of a byte into character
            char ch = (char) k;
            System.out.println("ASCII value of Character is:" + k + "; Special character is: " + ch);
        }
    }
}
```

Output:

```
ASCII value of Character is:35; Special character is: #
ASCII value of Character is:36; Special character is: $
```

ASCII value of Character is:37; Special character is: %

ASCII value of Character is:38; Special character is: &

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

learning

Java DataOutputStream Class

Java DataOutputStream **class** allows an application to write primitive **Java** data types to the output stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

Java DataOutputStream class declaration

Let's see the declaration for java.io.DataOutputStream class:

```
public class DataOutputStream extends FilterOutputStream implements DataOutput
```

Java DataOutputStream class methods

Method	Description
int size()	It is used to return the number of bytes written to the data output stream.
void write(int b)	It is used to write the specified byte to the underlying output stream.
void write(byte[] b, int off, int len)	It is used to write len bytes of data to the output stream.
void writeBoolean(boolean v)	It is used to write Boolean to the output stream as a 1-byte value.
void writeChar(int v)	It is used to write char to the output stream as a 2-byte value.
void writeChars(String s)	It is used to write string to the output stream as a sequence of characters.
void writeByte(int v)	It is used to write a byte to the output stream as a 1-byte value.
void writeBytes(String s)	It is used to write string to the output stream as a sequence of bytes.
void writeInt(int v)	It is used to write an int to the output stream

void writeShort(int v)	It is used to write a short to the output stream.
void writeShort(int v)	It is used to write a short to the output stream.
void writeLong(long v)	It is used to write a long to the output stream.
void writeUTF(String str)	It is used to write a string to the output stream using UTF-8 encoding in portable manner.
void flush()	It is used to flushes the data output stream.

Example of DataOutputStream class

In this example, we are writing the data to a text file **testout.txt** using DataOutputStream class.

```
package com.javatpoint;

import java.io.*;
public class OutputExample {
    public static void main(String[] args) throws IOException {
        FileOutputStream file = new FileOutputStream(D:\\testout.txt);
        DataOutputStream data = new DataOutputStream(file);
        data.writeInt(65);
        data.flush();
        data.close();
        System.out.println("Success...");
    }
}
```

Output:

Success...

testout.txt:

A

Java DataInputStream Class

Java DataInputStream **class** allows an application to read primitive data from the input stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

Java DataInputStream class declaration

Let's see the declaration for java.io.DataInputStream class:

```
public class DataInputStream extends FilterInputStream implements DataInput
```

Java DataInputStream class Methods

Method	Description
int read(byte[] b)	It is used to read the number of bytes from the input stream.
int read(byte[] b, int off, int len)	It is used to read len bytes of data from the input stream.
int readInt()	It is used to read input bytes and return an int value.
byte readByte()	It is used to read and return the one input byte.
char readChar()	It is used to read two input bytes and returns a char value.
double readDouble()	It is used to read eight input bytes and returns a double value.
boolean readBoolean()	It is used to read one input byte and return true if byte is non zero, false if byte is zero.
int skipBytes(int x)	It is used to skip over x bytes of data from the input stream.
String readUTF()	It is used to read a string that has been encoded using the UTF-8 format.

void readFully(byte[] b)	It is used to read bytes from the input stream and store them into the buffer array .
void readFully(byte[] b, int off, int len)	It is used to read len bytes from the input stream.

Example of DataInputStream class

In this example, we are reading the data from the file testout.txt file.

```
package com.javatpoint;
import java.io.*;
public class DataStreamExample {
    public static void main(String[] args) throws IOException {
        InputStream input = new FileInputStream("D:\\testout.txt");
        DataInputStream inst = new DataInputStream(input);
        int count = input.available();
        byte[] ary = new byte[count];
        inst.read(ary);
        for (byte bt : ary) {
            char k = (char) bt;
            System.out.print(k + "-");
        }
    }
}
```

Here, we are assuming that you have following data in "**testout.txt**" file:

JAVA

Output:

J-A-V-A

Java FilterOutputStream Class

Java FilterOutputStream class implements the OutputStream [class](#). It provides different sub classes such as [BufferedOutputStream](#) and [DataOutputStream](#) to provide additional functionality. So it is less used individually.

Java FilterOutputStream class declaration

Let's see the declaration for java.io.FilterOutputStream class:

```
public class FilterOutputStream extends OutputStream
```

Java FilterOutputStream class Methods

Method	Description
void write(int b)	It is used to write the specified byte to the output stream.
void write(byte[] ary)	It is used to write ary.length byte to the output stream.
void write(byte[] b, int off, int len)	It is used to write len bytes from the offset off to the output stream.
void flush()	It is used to flushes the output stream.
void close()	It is used to close the output stream.

Example of FilterOutputStream class

```
import java.io.*;
public class FilterExample {
    public static void main(String[] args) throws IOException {
        File data = new File("D:\\testout.txt");
        FileOutputStream file = new FileOutputStream(data);
        FilterOutputStream filter = new FilterOutputStream(file);
        String s="Welcome to javaTpoint.";
        byte b[]={s.getBytes()};
        filter.write(b);
    }
}
```

```
filter.flush();
filter.close();
file.close();
System.out.println("Success...");
}

}
```

Output:

```
Success...
```

testout.txt

```
Welcome to javaTpoint.
```

← Prev

Next →

AD

 YouTube For Videos Join Our Youtube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java FilterInputStream Class

Java FilterInputStream class implements the InputStream. It contains different sub classes as **BufferedInputStream**, **DataInputStream** for providing additional functionality. So it is less used individually.

Java FilterInputStream class declaration

Let's see the declaration for java.io.FilterInputStream class

```
public class FilterInputStream extends InputStream
```

Java FilterInputStream class Methods

Method	Description
int available()	It is used to return an estimate number of bytes that can be read from the input stream.
int read()	It is used to read the next byte of data from the input stream.
int read(byte[] b)	It is used to read up to byte.length bytes of data from the input stream.
long skip(long n)	It is used to skip over and discards n bytes of data from the input stream.
boolean markSupported()	It is used to test if the input stream support mark and reset method.
void mark(int readlimit)	It is used to mark the current position in the input stream.
void reset()	It is used to reset the input stream.
void close()	It is used to close the input stream.

Example of FilterInputStream class

```
import java.io.*;
public class FilterExample {
    public static void main(String[] args) throws IOException {
```

```
File data = new File("D:\\testout.txt");
FileInputStream file = new FileInputStream(data);
FilterInputStream filter = new BufferedInputStream(file);
int k =0;
while((k=filter.read())!=-1){
    System.out.print((char)k);
}
file.close();
filter.close();
}
```

Here, we are assuming that you have following data in "**testout.txt**" file:

```
Welcome to javatpoint
```

Output:

```
Welcome to javatpoint
```

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Java - ObjectOutputStream

ObjectOutputStream act as a **Serialization** descriptor for class. This **class** contains the name and serialVersionUID of the class.

Fields

Modifier and Type	Field	Description
static ObjectStreamField[]	NO_FIELDS	serialPersistentFields value indicating no serializable fields

Methods

Modifier and Type	Method	Description
Class<?>	forClass()	It returns the class in the local VM that this version is mapped to.
ObjectStreamField	getField(String name)	It gets the field of this class by name.
ObjectStreamField[]	getFields()	It returns an array of the fields of this serialization class.
String	getName()	It returns the name of the class described by this descriptor.
long	getSerialVersionUID()	It returns the serialVersionUID for this class.
Static ObjectOutputStream	lookup(Class<?> cl)	It finds the descriptor for a class that can be serialized.
Static ObjectOutputStream	lookupAny(Class<?> cl)	It returns the descriptor for any class, regardless of whether it implements Serializable.
String	toString()	It returns a string describing this ObjectOutputStream.

Example

```
import java.io.ObjectStreamClass;
import java.util.Calendar;

public class ObjectStreamClassExample {
    public static void main(String[] args) {

        // create a new object stream class for Integers
        ObjectStreamClass osc = ObjectStreamClass.lookup(SmartPhone.class);

        // get the value field from ObjectStreamClass for integers
        System.out.println("+" + osc.getField("price"));

        // create a new object stream class for Calendar
        ObjectStreamClass osc2 = ObjectStreamClass.lookup(String.class);

        // get the Class instance for osc2
        System.out.println("+" + osc2.getField("hash"));

    }
}
```

Output:

```
I price
null
```

← Prev

Next →

Java ObjectOutputStream class

A description of a Serializable field from a **Serializable** class. An **array** of ObjectStreamFields is used to declare the Serializable fields of a class.

The `java.io.ObjectStreamClass.getField(String name)` method gets the field of this class by name.

Constructor

Constructor	Description
<code>ObjectStreamField(String name, Class<?> type)</code>	It creates a Serializable field with the specified type.
<code>ObjectStreamField(String name, Class<?> type, boolean unshared)</code>	It creates an ObjectStreamField representing a serializable field with the given name and type.

Methods

Modifier and Type	Method	Description
int	<code>compareTo(Object obj)</code>	It compares this field with another ObjectStreamField.
String	<code>getName()</code>	It gets the name of this field.
int	<code>GetOffset()</code>	Offset of field within instance data.
<code>Class<?></code>	<code>getType()</code>	It get the type of the field.
char	<code>getTypeCode()</code>	It returns character encoding of field type.
String	<code>getTypeString()</code>	It return the JVM type signature.
boolean	<code>isPrimitive()</code>	It return true if this field has a primitive type.
boolean	<code>isUnshared()</code>	It returns boolean value indicating whether or not the serializable field represented by this ObjectStreamField instance is unshared.

protected void	setOffset(int offset)	Offset within instance data.
String	toString()	It return a string that describes this field.

public char getTypeCode()

Returns character encoding of field type. The encoding is as follows:

B	byte
C	char
D	double
F	float
I	int
J	long
L	class or interface
S	short
Z	boolean
[array

Returns:

the typecode of the serializable field

Example:

```
import java.io.ObjectStreamClass;
import java.util.Calendar;

public class ObjectStreamClassExample {
    public static void main(String[] args) {

        // create a new object stream class for Integers
        ObjectStreamClass osc = ObjectStreamClass.lookup(String.class);
```

```
// get the value field from ObjectStreamClass for integers  
System.out.println(" " + osc.getField("value"));  
  
// create a new object stream class for Calendar  
ObjectStreamClass osc2 = ObjectStreamClass.lookup(Calendar.class);  
  
// get the Class instance for osc2  
System.out.println(" " + osc2.getField("isTimeSet"));  
  
}  
}
```

Output:

```
I value  
Z isTimeSet
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: Join Now

Java Console Class

The Java Console class is used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The `java.io.Console` class is attached with system console internally. The `Console` class is introduced since 1.5.

Let's see a simple example to read text from console.

```
String text=System.console().readLine();
System.out.println("Text is: "+text);
```

Java Console class declaration

Let's see the declaration for `java.io.Console` class:

```
public final class Console extends Object implements Flushable
```

Java Console class methods

Method	Description
<code>Reader reader()</code>	It is used to retrieve the reader <code>object</code> associated with the console
<code>String readLine()</code>	It is used to read a single line of text from the console.
<code>String readLine(String fmt, Object... args)</code>	It provides a formatted prompt then reads the single line of text from the console.
<code>char[] readPassword()</code>	It is used to read password that is not being displayed on the console.
<code>char[] readPassword(String fmt, Object... args)</code>	It provides a formatted prompt then reads the password that is not being displayed on the console.

Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

How to get the object of Console

System class provides a static method `console()` that returns the **singleton** instance of `Console` class.

```
public static Console console()
```

Let's see the code to get the instance of `Console` class.

```
Console c=System.console();
```



Java Console Example

```
import java.io.Console;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);
}
}
```

Output

```
Enter your name: Nakul Jain  
Welcome Nakul Jain
```

Java Console Example to read password

```
import java.io.Console;  
class ReadPasswordTest{  
public static void main(String args[]){  
Console c=System.console();  
System.out.println("Enter password: ");  
char[] ch=c.readPassword();  
String pass=String.valueOf(ch);//converting char array into string  
System.out.println("Password is: "+pass);  
}  
}
```

Output

```
Enter password:  
Password is: 123
```

← Prev

Next →

Java FilePermission Class

Java FilePermission class contains the permission related to a directory or **file**. All the permissions are related with path. The path can be of two types:

- 1) **D:\\IO\\-**: It indicates that the permission is associated with all sub directories and files recursively.
- 2) **D:\\IO***: It indicates that the permission is associated with all directory and files within this directory excluding sub directories.

Java FilePermission class declaration

Let's see the declaration for Java.io.FilePermission class:

```
public final class FilePermission extends Permission implements Serializable
```

Methods of FilePermission class

Method	Description
ByteArrayOutputStream()	Creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int size)	Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Java FilePermission class methods

Method	Description
int hashCode()	It is used to return the hash code value of an object .
String getActions()	It is used to return the "canonical string representation" of an action.
boolean equals(Object obj)	It is used to check the two FilePermission objects for equality.

boolean implies(Permission p)	It is used to check the FilePermission object for the specified permission.
PermissionCollection newPermissionCollection()	It is used to return the new PermissionCollection object for storing the FilePermission object.



Java FilePermission Example

Let's see the simple example in which permission of a directory path is granted with read permission and a file of this directory is granted for write permission.

```
package com.javatpoint;

import java.io.*;
import java.security.PermissionCollection;
public class FilePermissionExample{
    public static void main(String[] args) throws IOException {
        String srg = "D:\\IO Package\\java.txt";
        FilePermission file1 = new FilePermission("D:\\IO Package\\-", "read");
        PermissionCollection permission = file1.newPermissionCollection();
        permission.add(file1);
        FilePermission file2 = new FilePermission(srg, "write");
        permission.add(file2);
        if(permission.implies(new FilePermission(srg, "read,write"))){
            System.out.println("Read, Write permission is granted for the path "+srg );
        }else {
            System.out.println("No Read, Write permission is granted for the path "+srg);
        }
    }
}
```

Output

```
Read, Write permission is granted for the path D:\\IO Package\\java.txt
```

Java Writer

It is an **abstract** class for writing to character streams. The methods that a subclass must implement are `write(char[], int, int)`, `flush()`, and `close()`. Most subclasses will override some of the methods defined here to provide higher efficiency, functionality or both.

Fields

Modifier and Type	Field	Description
protected Object	lock	The object used to synchronize operations on this stream.

Constructor

Modifier	Constructor	Description
protected	<code>Writer()</code>	It creates a new character-stream writer whose critical sections will synchronize on the writer itself.
protected	<code>Writer(Object lock)</code>	It creates a new character-stream writer whose critical sections will synchronize on the given object .

Methods

Modifier and Type	Method	Description
Writer	<code>append(char c)</code>	It appends the specified character to this writer.
Writer	<code>append(CharSequence csq)</code>	It appends the specified character sequence to this writer
Writer	<code>append(CharSequence csq, int start, int end)</code>	It appends a subsequence of the specified character sequence to this writer.
abstract void	<code>close()</code>	It closes the stream, flushing it first.
abstract void	<code>flush()</code>	It flushes the stream.
void	<code>write(char[] cbuf)</code>	It writes an array of characters.

abstract void	write(char[] cbuf, int off, int len)	It writes a portion of an array of characters.
void	write(int c)	It writes a single character.
void	write(String str)	It writes a string .
void	write(String str, int off, int len)	It writes a portion of a string.

Java Writer Example

```

import java.io.*;
public class WriterExample {
    public static void main(String[] args) {
        try {
            Writer w = new FileWriter("output.txt");
            String content = "I love my country";
            w.write(content);
            w.close();
            System.out.println("Done");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Output:

Done

output.txt:

I love my country

← Prev

Next →

Java Reader

Java Reader is an **abstract class** for reading character **streams**. The only methods that a subclass must implement are `read(char[], int, int)` and `close()`. Most subclasses, however, will **override** some of the methods to provide higher efficiency, additional functionality, or both.

Some of the implementation **class** are `BufferedReader`, `CharArrayReader`, `FilterReader`, `InputStreamReader`, `PipedReader`, `StringReader`

Fields

Modifier and Type	Field	Description
protected Object	lock	The object used to synchronize operations on this stream.

Constructor

Modifier	Constructor	Description
protected	<code>Reader()</code>	It creates a new character-stream reader whose critical sections will synchronize on the reader itself.
protected	<code>Reader(Object lock)</code>	It creates a new character-stream reader whose critical sections will synchronize on the given object.

Methods

Modifier and Type	Method	Description
abstract void	<code>close()</code>	It closes the stream and releases any system resources associated with it.
void	<code>mark(int readAheadLimit)</code>	It marks the present position in the stream.
boolean	<code>markSupported()</code>	It tells whether this stream supports the <code>mark()</code> operation.
int	<code>read()</code>	It reads a single character.
int	<code>read(char[] cbuf)</code>	It reads characters into an array .

abstract int	read(char[] cbuf, int off, int len)	It reads characters into a portion of an array.
int	read(CharBuffer target)	It attempts to read characters into the specified character buffer.
boolean	ready()	It tells whether this stream is ready to be read.
void	reset()	It resets the stream.
long	skip(long n)	It skips characters.

Example

```

import java.io.*;
public class ReaderExample {
    public static void main(String[] args) {
        try {
            Reader reader = new FileReader("file.txt");
            int data = reader.read();
            while (data != -1) {
                System.out.print((char) data);
                data = reader.read();
            }
            reader.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}

```

file.txt:

I love my country

Output:

I love my country

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

 [Tumblr tutorial](#)

Tumblr

 [React tutorial](#)

ReactJS

 [Regex tutorial](#)

Regex

 [Reinforcement learning tutorial](#)

learning

Java FileWriter Class

Java FileWriter class is used to write character-oriented data to a **file**. It is character-oriented class which is used for file handling in **java**.

Unlike FileOutputStream class, you don't need to convert string into byte **array** because it provides method to write string directly.

Java FileWriter class declaration

Let's see the declaration for Java.io.FileWriter class:

```
public class FileWriter extends OutputStreamWriter
```

Constructors of FileWriter class

Constructor	Description
FileWriter(String file)	Creates a new file. It gets file name in string .
FileWriter(File file)	Creates a new file. It gets file name in File object .

Methods of FileWriter class

Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Java FileWriter Example

In this example, we are writing the data in the file testout.txt using Java FileWriter class.

```
package com.javatpoint;  
import java.io.FileWriter;  
public class FileWriterExample {  
    public static void main(String args[]){  
        try{  
            FileWriter fw=new FileWriter("D:\\testout.txt");  
            fw.write("Welcome to javaTpoint.");  
            fw.close();  
        }catch(Exception e){System.out.println(e);}  
        System.out.println("Success...");  
    }  
}
```

Output:

```
Success...
```

testout.txt:

```
Welcome to javaTpoint.
```

← Prev

Next →

Java FileReader Class

Java FileReader class is used to read data from the file. It returns data in byte format like [FileInputStream](#) class.

It is character-oriented class which is used for [file](#) handling in [java](#).

Java FileReader class declaration

Let's see the declaration for Java.io.FileReader class:

```
public class FileReader extends InputStreamReader
```

Constructors of FileReader class

Constructor	Description
FileReader(String file)	It gets filename in string . It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException .
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException .

Methods of FileReader class

Method	Description
int read()	It is used to return a character in ASCII form. It returns -1 at the end of file.
void close()	It is used to close the FileReader class.



Java FileReader Example

In this example, we are reading the data from the text file **testout.txt** using Java FileReader class.

```
package com.javatpoint;

import java.io.FileReader;
public class FileReaderExample {
    public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        int i;
        while((i=fr.read())!=-1)
            System.out.print((char)i);
        fr.close();
    }
}
```

Here, we are assuming that you have following data in "testout.txt" file:

```
Welcome to javaTpoint.
```

Output:

```
Welcome to javaTpoint.
```

← Prev

Next →

AD



For Videos Join Our Youtube Channel: [Join Now](#)

Java BufferedWriter Class

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits **Writer** class. The buffering characters are used for providing the efficient writing of single **arrays**, characters, and **strings**.

Class declaration

Let's see the declaration for Java.io.BufferedWriter class:

```
public class BufferedWriter extends Writer
```

Class constructors

Constructor	Description
BufferedWriter(Writer wrt)	It is used to create a buffered character output stream that uses the default size for an output buffer.
BufferedWriter(Writer wrt, int size)	It is used to create a buffered character output stream that uses the specified size for an output buffer.

Class methods

Method	Description
void newLine()	It is used to add a new line by writing a line separator.
void write(int c)	It is used to write a single character.
void write(char[] cbuf, int off, int len)	It is used to write a portion of an array of characters.
void write(String s, int off, int len)	It is used to write a portion of a string.
void flush()	It is used to flushes the input stream.
void close()	It is used to closes the input stream

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

Example of Java BufferedWriter

Let's see the simple example of writing the data to a text file **testout.txt** using Java BufferedWriter.

```
package com.javatpoint;  
import java.io.*;  
public class BufferedWriterExample {  
    public static void main(String[] args) throws Exception {  
        FileWriter writer = new FileWriter("D:\\testout.txt");  
        BufferedWriter buffer = new BufferedWriter(writer);  
        buffer.write("Welcome to javaTpoint.");  
        buffer.close();  
        System.out.println("Success");  
    }  
}
```

Output:

```
success
```

testout.txt:

```
Welcome to javaTpoint.
```

← Prev

Next →

Java BufferedReader Class

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits [Reader class](#).

Java BufferedReader class declaration

Let's see the declaration for Java.io.BufferedReader class:

```
public class BufferedReader extends Reader
```

Java BufferedReader class constructors

Constructor	Description
BufferedReader(Reader rd)	It is used to create a buffered character input stream that uses the default size for an input buffer.
BufferedReader(Reader rd, int size)	It is used to create a buffered character input stream that uses the specified size for an input buffer.

Java BufferedReader class methods

Method	Description
int read()	It is used for reading a single character.
int read(char[] cbuf, int off, int len)	It is used for reading characters into a portion of an array .
boolean markSupported()	It is used to test the input stream support for the mark and reset method.
String readLine()	It is used for reading a line of text.
boolean ready()	It is used to test whether the input stream is ready to be read.

long skip(long n)	It is used for skipping the characters.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readAheadLimit)	It is used for marking the present position in a stream.
void close()	It closes the input stream and releases any of the system resources associated with the stream.

AD



Java BufferedReader Example

In this example, we are reading the data from the text file **testout.txt** using Java BufferedReader class.

```
package com.javatpoint;
import java.io.*;
public class BufferedReaderExample {
    public static void main(String args[]) throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        BufferedReader br=new BufferedReader(fr);

        int i;
        while((i=br.read())!=-1){
            System.out.print((char)i);
        }
        br.close();
        fr.close();
    }
}
```

Here, we are assuming that you have following data in "testout.txt" file:

Welcome to javaTpoint.

Output:

```
Welcome to javaTpoint.
```

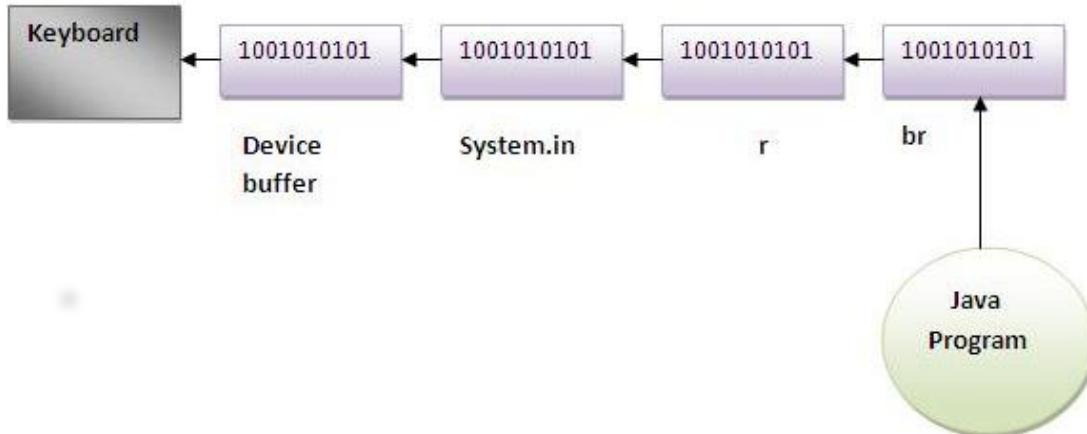
Reading data from console by InputStreamReader and BufferedReader

In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

```
package com.javatpoint;
import java.io.*;
public class BufferedReaderExample{
    public static void main(String args[]) throws Exception{
        InputStreamReader r=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(r);
        System.out.println("Enter your name");
        String name=br.readLine();
        System.out.println("Welcome "+name);
    }
}
```

Output:

```
Enter your name
Nakul Jain
Welcome Nakul Jain
```



Another example of reading data from console until user writes stop

In this example, we are reading and printing the data until the user prints stop.

```
package com.javatpoint;  
import java.io.*;  
public class BufferedReaderExample{  
    public static void main(String args[])throws Exception{  
        InputStreamReader r=new InputStreamReader(System.in);  
        BufferedReader br=new BufferedReader(r);  
        String name="";  
        while(!name.equals("stop")){  
            System.out.println("Enter data: ");  
            name=br.readLine();  
            System.out.println("data is: "+name);  
        }  
        br.close();  
        r.close();  
    }  
}
```

Output:

```
Enter data: Nakul  
data is: Nakul  
Enter data: 12  
data is: 12  
Enter data: stop  
data is: stop
```

← Prev

Next →

Java CharArrayReader Class

The CharArrayReader is composed of two words: CharArray and Reader. The CharArrayReader class is used to read character **array** as a reader (stream). It inherits **Reader** class.

Java CharArrayReader class declaration

Let's see the declaration for Java.io.CharArrayReader **class**:

```
public class CharArrayReader extends Reader
```

Java CharArrayReader class methods

Method	Description
int read()	It is used to read a single character
int read(char[] b, int off, int len)	It is used to read characters into the portion of an array.
boolean ready()	It is used to tell whether the stream is ready to read.
boolean markSupported()	It is used to tell whether the stream supports mark() operation.
long skip(long n)	It is used to skip the character in the input stream.
void mark(int readAheadLimit)	It is used to mark the present position in the stream.
void reset()	It is used to reset the stream to a most recent mark.
void close()	It is used to closes the stream.

Example of CharArrayReader Class:

Let's see the simple example to read a character using Java CharArrayReader class.

```
package com.javatpoint;

import java.io.CharArrayReader;
public class CharArrayExample{
```

```
public static void main(String[] args) throws Exception {
    char[] ary = { 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't' };
    CharArrayReader reader = new CharArrayReader(ary);
    int k = 0;
    // Read until the end of a file
    while ((k = reader.read()) != -1) {
        char ch = (char) k;
        System.out.print(ch + " : ");
        System.out.println(k);
    }
}
```

Output

```
j : 106
a : 97
v : 118
a : 97
t : 116
p : 112
o : 111
i : 105
n : 110
t : 116
```

← Prev

Next →

Java CharArrayWriter Class

The CharArrayWriter class can be used to write common data to multiple files. This class inherits [Writer](#) class. Its buffer automatically grows when data is written in this stream. Calling the `close()` method on this [object](#) has no effect.

Java CharArrayWriter class declaration

Let's see the declaration for `Java.io.CharArrayWriter` class:

```
public class CharArrayWriter extends Writer
```

Java CharArrayWriter class Methods

Method	Description
<code>int size()</code>	It is used to return the current size of the buffer.
<code>char[] toCharArray()</code>	It is used to return the copy of an input data.
<code>String toString()</code>	It is used for converting an input data to a string .
<code>CharArrayWriter append(char c)</code>	It is used to append the specified character to the writer.
<code>CharArrayWriter append(CharSequence csq)</code>	It is used to append the specified character sequence to the writer.
<code>CharArrayWriter append(CharSequence csq, int start, int end)</code>	It is used to append the subsequence of a specified character to the writer.
<code>void write(int c)</code>	It is used to write a character to the buffer.
<code>void write(char[] c, int off, int len)</code>	It is used to write a character to the buffer.
<code>void write(String str, int off, int len)</code>	It is used to write a portion of string to the buffer.
<code>void writeTo(Writer out)</code>	It is used to write the content of buffer to different character stream.

void flush()	It is used to flush the stream.
void reset()	It is used to reset the buffer.
void close()	It is used to close the stream.

Example of CharArrayWriter Class:

In this example, we are writing a common data to 4 files a.txt, b.txt, c.txt and d.txt.

```
package com.javatpoint;

import java.ioCharArrayWriter;
import java.io.FileWriter;
public class CharArrayWriterExample {
    public static void main(String args[])throws Exception{
        CharArrayWriter out=new CharArrayWriter();
        out.write("Welcome to javaTpoint");
        FileWriter f1=new FileWriter("D:\\a.txt");
        FileWriter f2=new FileWriter("D:\\b.txt");
        FileWriter f3=new FileWriter("D:\\c.txt");
        FileWriter f4=new FileWriter("D:\\d.txt");
        out.writeTo(f1);
        out.writeTo(f2);
        out.writeTo(f3);
        out.writeTo(f4);
        f1.close();
        f2.close();
        f3.close();
        f4.close();
        System.out.println("Success... ");
    }
}
```

Output

Success...

After executing the program, you can see that all files have common data: Welcome to javaTpoint.

a.txt:

```
Welcome to javaTpoint
```

b.txt:

```
Welcome to javaTpoint
```

c.txt:

```
Welcome to javaTpoint
```

d.txt:

```
Welcome to javaTpoint
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java PrintStream Class

The PrintStream class provides methods to write data to another stream. The PrintStream **class** automatically flushes the data so there is no need to call `flush()` method. Moreover, its methods don't throw `IOException`.

Class declaration

Let's see the declaration for `Java.io.PrintStream` class:

```
public class PrintStream extends FilterOutputStream implements Closeable, Appendable
```

Methods of PrintStream class

Method	Description
<code>void print(boolean b)</code>	It prints the specified boolean value.
<code>void print(char c)</code>	It prints the specified char value.
<code>void print(char[] c)</code>	It prints the specified character array values.
<code>void print(int i)</code>	It prints the specified int value.
<code>void print(long l)</code>	It prints the specified long value.
<code>void print(float f)</code>	It prints the specified float value.
<code>void print(double d)</code>	It prints the specified double value.
<code>void print(String s)</code>	It prints the specified string value.
<code>void print(Object obj)</code>	It prints the specified object value.
<code>void println(boolean b)</code>	It prints the specified boolean value and terminates the line.
<code>void println(char c)</code>	It prints the specified char value and terminates the line.

void println(char[] c)	It prints the specified character array values and terminates the line.
void println(int i)	It prints the specified int value and terminates the line.
void println(long l)	It prints the specified long value and terminates the line.
void println(float f)	It prints the specified float value and terminates the line.
void println(double d)	It prints the specified double value and terminates the line.
void println(String s)	It prints the specified string value and terminates the line.
void println(Object obj)	It prints the specified object value and terminates the line.
void println()	It terminates the line only.
void printf(Object format, Object... args)	It writes the formatted string to the current stream.
void printf(Locale l, Object format, Object... args)	It writes the formatted string to the current stream.
void format(Object format, Object... args)	It writes the formatted string to the current stream using specified format.
void format(Locale l, Object format, Object... args)	It writes the formatted string to the current stream using specified format.

Example of java PrintStream class

In this example, we are simply printing integer and string value.

```
package com.javatpoint;

import java.io.FileOutputStream;
import java.io.PrintStream;
public class PrintStreamTest{
```

```
public static void main(String args[])throws Exception{
    FileOutputStream fout=new FileOutputStream("D:\\testout.txt ");
    PrintStream pout=new PrintStream(fout);
    pout.println(2016);
    pout.println("Hello Java");
    pout.println("Welcome to Java");
    pout.close();
    fout.close();
    System.out.println("Success?");
}
```

Output

```
Success...
```

The content of a text file **testout.txt** is set with the below data

```
2016
Hello Java
Welcome to Java
```



Example of printf() method using java PrintStream class:

Let's see the simple example of printing integer value by format specifier using **printf()** method of **java.io.PrintStream** class.

```
class PrintStreamTest{
    public static void main(String args[]){
        int a=19;
        System.out.printf("%d",a); //Note: out is the object of printstream
    }
}
```

{}

Output

19

[← Prev](#)[Next →](#)

AD

 For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 Splunk tutorial

Splunk

 SPSS tutorial

SPSS



Swagger tutorial

Swagger

 T-SQL tutorial

Transact-SQL

Java PrintWriter class

Java PrintWriter class is the implementation of **Writer** class. It is used to print the formatted representation of **objects** to the text-output stream.

Class declaration

Let's see the declaration for Java.io.PrintWriter class:

```
public class PrintWriter extends Writer
```

Methods of PrintWriter class

Method	Description
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.
PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.

void flush()	It is used to flushes the stream.
void close()	It is used to close the stream.

Java PrintWriter Example

Let's see the simple example of writing the data on a **console** and in a **text file testout.txt** using Java PrintWriter class.

```
package com.javatpoint;

import java.io.File;
import java.io.PrintWriter;
public class PrintWriterExample {
    public static void main(String[] args) throws Exception {
        //Data to write on Console using PrintWriter
        PrintWriter writer = new PrintWriter(System.out);
        writer.write("Javatpoint provides tutorials of all technology.");
        writer.flush();
        writer.close();
        //Data to write in File using PrintWriter
        PrintWriter writer1 =null;
        writer1 = new PrintWriter(new File("D:\\testout.txt"));
        writer1.write("Like Java, Spring, Hibernate, Android, PHP etc.");
        writer1.flush();
        writer1.close();
    }
}
```

Output

```
Javatpoint provides tutorials of all technology.
```

The content of a text file **testout.txt** is set with the data **Like Java, Spring, Hibernate, Android, PHP etc.**

Java OutputStreamWriter

OutputStreamWriter is a **class** which is used to convert character stream to byte stream, the characters are encoded into byte using a specified charset. write() method calls the encoding converter which converts the character into bytes. The resulting bytes are then accumulated in a buffer before being written into the underlying output stream. The characters passed to write() methods are not buffered. We optimize the performance of OutputStreamWriter by using it with in a BufferedWriter so that to avoid frequent converter invocation.

Constructor

Constructor	Description
OutputStreamWriter(OutputStream out)	It creates an OutputStreamWriter that uses the default character encoding.
OutputStreamWriter(OutputStream out, Charset cs)	It creates an OutputStreamWriter that uses the given charset.
OutputStreamWriter(OutputStream out, CharsetEncoder enc)	It creates an OutputStreamWriter that uses the given charset encoder.
OutputStreamWriter(OutputStream out, String charsetName)	It creates an OutputStreamWriter that uses the named charset.

Methods

Modifier and Type	Method	Description
void	close()	It closes the stream, flushing it first.
void	flush()	It flushes the stream.
String	getEncoding()	It returns the name of the character encoding being used by this stream.
void	write(char[] cbuf, int off, int len)	It writes a portion of an array of characters.
void	write(int c)	It writes a single character.

void	write(String str, int off, int len)	It writes a portion of a string .
------	-------------------------------------	--

Example

```
public class OutputStreamWriterExample {  
    public static void main(String[] args) {  
  
        try {  
            OutputStream outputStream = new FileOutputStream("output.txt");  
            Writer outputStreamWriter = new OutputStreamWriter(outputStream);  
  
            outputStreamWriter.write("Hello World");  
  
            outputStreamWriter.close();  
        } catch (Exception e) {  
            e.getMessage();  
        }  
    }  
}
```

Output:

```
output.txt file will contains text "Hello World"
```

← Prev

Next →

Java InputStreamReader

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Constructor

Constructor name	Description
InputStreamReader(InputStream in)	It creates an InputStreamReader that uses the default charset.
InputStreamReader(InputStream in, Charset cs)	It creates an InputStreamReader that uses the given charset.
InputStreamReader(InputStream in, CharsetDecoder dec)	It creates an InputStreamReader that uses the given charset decoder.
InputStreamReader(InputStream in, String charsetName)	It creates an InputStreamReader that uses the named charset.

Method

Modifier and Type	Method	Description
void	close()	It closes the stream and releases any system resources associated with it.
String	getEncoding()	It returns the name of the character encoding being used by this stream.
int	read()	It reads a single character.
int	read(char[] cbuf, int offset, int length)	It reads characters into a portion of an array.
boolean	ready()	It tells whether this stream is ready to be read.

Example

```
public class InputStreamReaderExample {  
    public static void main(String[] args) {  
        try {  
            InputStream stream = new FileInputStream("file.txt");  
            Reader reader = new InputStreamReader(stream);  
            int data = reader.read();  
            while (data != -1) {  
                System.out.print((char) data);  
                data = reader.read();  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

I love my country

The file.txt contains text "I love my country" the InputStreamReader reads Character by character from the file

← Prev

Next →

Java PushbackInputStream Class

Java PushbackInputStream **class** overrides InputStream and provides extra functionality to another input stream. It can unread a byte which is already read and push back one byte.

Class declaration

Let's see the declaration for java.io.PushbackInputStream class:

```
public class PushbackInputStream extends FilterInputStream
```

Class Methods

It is used to test if the input stream support mark and reset method.

Method	Description
int available()	It is used to return the number of bytes that can be read from the input stream.
int read()	It is used to read the next byte of data from the input stream.
boolean markSupported()	
void mark(int readlimit)	It is used to mark the current position in the input stream.
long skip(long x)	It is used to skip over and discard x bytes of data.
void unread(int b)	It is used to pushes back the byte by copying it to the pushback buffer.
void unread(byte[] b)	It is used to pushes back the array of byte by copying it to the pushback buffer.
void reset()	It is used to reset the input stream.
void close()	It is used to close the input stream.

Example of PushbackInputStream class

```
import java.io.*;

public class InputStreamExample {

    public static void main(String[] args) throws Exception{
        String srg = "1##2#34##12";
        byte ary[] = srg.getBytes();
        ByteArrayInputStream array = new ByteArrayInputStream(ary);
        PushbackInputStream push = new PushbackInputStream(array);

        int i;

        while( (i = push.read())!= -1) {
            if(i == '#') {
                int j;
                if( (j = push.read()) == '#'){
                    System.out.print("##");
                }else {
                    push.unread(j);
                    System.out.print((char)i);
                }
            }else {
                System.out.print((char)i);
            }
        }
    }
}
```

Output:

```
1**2#34**#12
```

← Prev

Next →

Java PushbackReader Class

Java PushbackReader class is a character stream reader. It is used to pushes back a character into stream and overrides the FilterReader class.

Class declaration

Let's see the declaration for java.io.PushbackReader class:

```
public class PushbackReader extends FilterReader
```

Class Methods

Method	Description
int read()	It is used to read a single character.
void mark(int readAheadLimit)	It is used to mark the present position in a stream.
boolean ready()	It is used to tell whether the stream is ready to be read.
boolean markSupported()	It is used to tell whether the stream supports mark() operation.
long skip(long n)	It is used to skip the character.
void unread (int c)	It is used to pushes back the character by copying it to the pushback buffer.
void unread (char[] cbuf)	It is used to pushes back an array of character by copying it to the pushback buffer.
void reset()	It is used to reset the stream.
void close()	It is used to close the stream.

Example of PushbackReader class

```
import java.io.*;
public class ReaderExample{
```

```
public static void main(String[] args) throws Exception {
```

```
    char ary[] = {'1', '-', '2', '-', '3', '4', '-', '-', '5', '6'};
```

```
    CharArrayReader reader = new CharArrayReader(ary);
```

```
    PushbackReader push = new PushbackReader(reader);
```

```
    int i;
```

```
    while( (i = push.read())!= -1) {
```

```
        if(i == '-') {
```

```
            int j;
```

```
            if( (j = push.read()) == '-' ){
```

```
                System.out.print("#*");
```

```
            }else {
```

```
                push.unread(j); // push back single character
```

```
                System.out.print((char)i);
```

```
            }
```

```
        }else {
```

```
            System.out.print((char)i);
```

```
        }
```

```
}
```

```
}
```

Output

```
1#*2-34#*-56
```

← Prev

Next →

AD



Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



Splunk



SPSS

Swagger
tutorial
Swagger

Transact-SQL



Tumblr



ReactJS



Regex

Reinforcement
learning
Reinforcement
Learning

R Programming



RxJS

React Native
tutorialPython Design
Patterns

Java StringWriter Class

Java StringWriter class is a character stream that collects output from string buffer, which can be used to construct a [string](#). The StringWriter class inherits the [Writer](#) class.

In StringWriter class, system resources like [network sockets](#) and [files](#) are not used, therefore closing the StringWriter is not necessary.

Java StringWriter class declaration

Let's see the declaration for Java.io.StringWriter class:

```
public class StringWriter extends Writer
```

Methods of StringWriter class

Method	Description
void write(int c)	It is used to write the single character.
void write(String str)	It is used to write the string.
void write(String str, int off, int len)	It is used to write the portion of a string.
void write(char[] cbuf, int off, int len)	It is used to write the portion of an array of characters.
String toString()	It is used to return the buffer current value as a string.
StringBuffer getBuffer()	It is used to return the string buffer.
StringWriter append(char c)	It is used to append the specified character to the writer.
StringWriter append(CharSequence csq)	It is used to append the specified character sequence to the writer.
StringWriter append(CharSequence csq, int start, int end)	It is used to append the subsequence of specified character sequence to the writer.

void flush()	It is used to flush the stream.
void close()	It is used to close the stream.

Java StringWriter Example

Let's see the simple example of StringWriter using BufferedReader to read file data from the stream.

```
import java.io.*;
public class StringWriterExample {
    public static void main(String[] args) throws IOException {
        char[] ary = new char[512];
        StringWriter writer = new StringWriter();
        FileInputStream input = null;
        BufferedReader buffer = null;
        input = new FileInputStream("D://testout.txt");
        buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
        int x;
        while ((x = buffer.read(ary)) != -1) {
            writer.write(ary, 0, x);
        }
        System.out.println(writer.toString());
        writer.close();
        buffer.close();
    }
}
```

testout.txt:

Javatpoint provides tutorial in Java, Spring, Hibernate, Android, PHP etc.

Output:

Javatpoint provides tutorial in Java, Spring, Hibernate, Android, PHP etc.

Java StringReader Class

Java StringReader class is a character stream with string as a source. It takes an input string and changes it into character stream. It inherits Reader class.

In StringReader class, system resources like network sockets and files are not used, therefore closing the StringReader is not necessary.

Java StringReader class declaration

Let's see the declaration for Java.io.StringReader class:

```
public class StringReader extends Reader
```

Methods of StringReader class

Method	Description
int read()	It is used to read a single character.
int read(char[] cbuf, int off, int len)	It is used to read a character into a portion of an array.
boolean ready()	It is used to tell whether the stream is ready to be read.
boolean markSupported()	It is used to tell whether the stream support mark() operation.
long skip(long ns)	It is used to skip the specified number of character in a stream
void mark(int readAheadLimit)	It is used to mark the mark the present position in a stream.
void reset()	It is used to reset the stream.
void close()	It is used to close the stream.

Java StringReader Example

```
import java.io.StringReader;  
  
public class StringReaderExample {  
    public static void main(String[] args) throws Exception {  
        String srg = "Hello Java!! \nWelcome to Javatpoint.";  
        StringReader reader = new StringReader(srg);  
        int k=0;  
        while((k=reader.read())!=-1){  
            System.out.print((char)k);  
        }  
    }  
}
```

Output:

```
Hello Java!!  
Welcome to Javatpoint.
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java - PipedWriter

The PipedWriter class is used to write [java](#) pipe as a stream of characters. This class is used generally for writing text. Generally PipedWriter is connected to a [PipedReader](#) and used by different [threads](#).

Constructor

Constructor	Description
PipedWriter()	It creates a piped writer that is not yet connected to a piped reader.
PipedWriter(PipedReader snk)	It creates a piped writer connected to the specified piped reader.

Method

Modifier and Type	Method	Method
void	close()	It closes this piped output stream and releases any system resources associated with this stream.
void	connect(PipedReader snk)	It connects this piped writer to a receiver.
void	flush()	It flushes this output stream and forces any buffered output characters to be written out.
void	write(char[] cbuf, int off, int len)	It writes len characters from the specified character array starting at offset off to this piped output stream.
void	write(int c)	It writes the specified char to the piped output stream.

Example

```
import java.io.PipedReader;
```

```
import java.io.PipedWriter;

public class PipeReaderExample2 {
    public static void main(String[] args) {
        try {

            final PipedReader read = new PipedReader();
            final PipedWriter write = new PipedWriter(read);

            Thread readerThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        int data = read.read();
                        while (data != -1) {
                            System.out.print((char) data);
                            data = read.read();
                        }
                    } catch (Exception ex) {
                    }
                }
            });
            });

            Thread writerThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        write.write("I love my country\n".toCharArray());
                    } catch (Exception ex) {
                    }
                }
            });
            });

            readerThread.start();
            writerThread.start();

        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

```
}
```

Output:

```
I love my country
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

 [SPSS tutorial](#)

Java - PipedReader

The PipedReader class is used to read the contents of a pipe as a stream of characters. This [class](#) is used generally to read text.

PipedReader class must be connected to the same [PipedWriter](#) and are used by different [threads](#).

Constructor

Constructor	Description
PipedReader(int pipeSize)	It creates a PipedReader so that it is not yet connected and uses the specified pipe size for the pipe's buffer.
PipedReader(PipedWriter src)	It creates a PipedReader so that it is connected to the piped writer src.
PipedReader(PipedWriter src, int pipeSize)	It creates a PipedReader so that it is connected to the piped writer src and uses the specified pipe size for the pipe's buffer.
PipedReader()	It creates a PipedReader so that it is not yet connected.

Method

Modifier and Type	Method	Method
void	close()	It closes this piped stream and releases any system resources associated with the stream.
void	connect(PipedWriter src)	It causes this piped reader to be connected to the piped writer src.
int	read()	It reads the next character of data from this piped stream.
int	read(char[] cbuf, int off, int len)	It reads up to len characters of data from this piped stream into an array of characters.
boolean	ready()	It tells whether this stream is ready to be read.

Example

```
import java.io.PipedReader;
import java.io.PipedWriter;

public class PipeReaderExample2 {
    public static void main(String[] args) {
        try {

            final PipedReader read = new PipedReader();
            final PipedWriter write = new PipedWriter(read);

            Thread readerThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        int data = read.read();
                        while (data != -1) {
                            System.out.print((char) data);
                            data = read.read();
                        }
                    } catch (Exception ex) {
                    }
                }
            });
            });

            Thread writerThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        write.write("I love my country\n".toCharArray());
                    } catch (Exception ex) {
                    }
                }
            });
            });

            readerThread.start();
            writerThread.start();
        }
    }
}
```

```
    } catch (Exception ex) {  
        System.out.println(ex.getMessage());  
    }  
  
}  
}
```

Output:

```
I love my country
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Java FilterWriter

Java FilterWriter class is an abstract **class** which is used to write filtered character streams.

The sub class of the FilterWriter should override some of its methods and it may provide additional methods and fields also.

Fields

Modifier	Type	Field	Description
protected	Writer	out	The underlying character-output stream.

Constructors

Modifier	Constructor	Description
protected	FilterWriter(Writer out)	It creates InputStream class Object

Methods

Modifier and Type	Method	Description
void	close()	It closes the stream, flushing it first.
void	flush()	It flushes the stream.
void	write(char[] cbuf, int off, int len)	It writes a portion of an array of characters.
void	write(int c)	It writes a single character.
void	write(String str, int off, int len)	It writes a portion of a string .

FilterWriter Example

```
import java.io.*;
class CustomFilterWriter extends FilterWriter {
```

```
CustomFilterWriter(Writer out) {  
    super(out);  
}  
  
public void write(String str) throws IOException {  
    super.write(str.toLowerCase());  
}  
}  
  
public class FilterWriterExample {  
    public static void main(String[] args) {  
        try {  
            FileWriter fw = new FileWriter("Record.txt");  
            CustomFilterWriter filterWriter = new CustomFilterWriter(fw);  
            filterWriter.write("I LOVE MY COUNTRY");  
            filterWriter.close();  
            FileReader fr = new FileReader("record.txt");  
            BufferedReader bufferedReader = new BufferedReader(fr);  
            int k;  
            while ((k = bufferedReader.read()) != -1) {  
                System.out.print((char) k);  
            }  
            bufferedReader.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Output:

```
i love my country
```

While running the current program if the current working directory does not contain the file, a new file is created and CustomFileWriter will write the text "I LOVE MY COUNTRY" in lowercase to the file.

Java FilterReader

Java FilterReader is used to perform filtering operation on **reader** stream. It is an abstract class for reading filtered character streams.

The FilterReader provides default methods that passes all requests to the contained stream. Subclasses of FilterReader should override some of its methods and may also provide additional methods and fields.

Field

Modifier	Type	Field	Description
protected	Reader	in	The underlying character-input stream.

Constructors

Modifier	Constructor	Description
protected	FilterReader(Reader in)	It creates a new filtered reader.

Method

Modifier and Type	Method	Description
void	close()	It closes the stream and releases any system resources associated with it.
void	mark(int readAheadLimit)	It marks the present position in the stream.
boolean	markSupported()	It tells whether this stream supports the mark() operation.
boolean	ready()	It tells whether this stream is ready to be read.
int	read()	It reads a single character.
int	read(char[] cbuf, int off, int len)	It reads characters into a portion of an array.

void	reset()	It resets the stream.
long	skip(long n)	It skips characters.

Example

In this example, we are using "javaFile123.txt" file which contains "India is my country" text in it. Here, we are converting whitespace with question mark '?'.

```

import java.io.*;
class CustomFilterReader extends FilterReader {
    CustomFilterReader(Reader in) {
        super(in);
    }
    public int read() throws IOException {
        int x = super.read();
        if ((char) x == ' ')
            return ((int) '?');
        else
            return x;
    }
}
public class FilterReaderExample {
    public static void main(String[] args) {
        try {
            Reader reader = new FileReader("javaFile123.txt");
            CustomFilterReader fr = new CustomFilterReader(reader);
            int i;
            while ((i = fr.read()) != -1) {
                System.out.print((char) i);
            }
            fr.close();
            reader.close();
        } catch (Exception e) {
            e.getMessage();
        }
    }
}

```

{}

Output:

```
India?is?my?country
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



Splunk



SPSS



Swagger
tutorial
Swagger



Transact-SQL

Java File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Fields

Modifier	Type	Field	Description
static	String	pathSeparator	It is system-dependent path-separator character, represented as a string for convenience.
static	char	pathSeparatorChar	It is system-dependent path-separator character.
static	String	separator	It is system-dependent default name-separator character, represented as a string for convenience.
static	char	separatorChar	It is system-dependent default name-separator character.

Constructors

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Useful Methods

Modifier and Type	Method	Description
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.

File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

Java File Example 1

```

import java.io.*;
public class FileDemo {
    public static void main(String[] args) {

        try {
            File file = new File("javaFile123.txt");
            if (file.createNewFile()) {
                System.out.println("New File is created!");
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}

```

Output:

New File is created!

Java File Example 2

```
import java.io.*;
public class FileDemo2 {
    public static void main(String[] args) {

        String path = "";
        boolean bool = false;
        try {
            // creating new files
            File file = new File("testFile1.txt");
            file.createNewFile();
            System.out.println(file);
            // creating new canonical from file object
            File file2 = file.getCanonicalFile();
            // returns true if the file exists
            System.out.println(file2);
            bool = file2.exists();
            // returns absolute pathname
            path = file2.getAbsolutePath();
            System.out.println(bool);
            // if file exists
            if (bool) {
                // prints
                System.out.print(path + " Exists? " + bool);
            }
        } catch (Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

Output:

```
testFile1.txt
/home/Work/Project/File/testFile1.txt
```

```
true  
/home/Work/Project/File/testFile1.txt Exists? true
```

Java File Example 3

```
import java.io.*;  
  
public class FileExample {  
  
    public static void main(String[] args) {  
  
        File f=new File("/Users/sonoojaiswal/Documents");  
  
        String filenames[]=f.list();  
  
        for(String filename:filenames){  
            System.out.println(filename);  
        }  
    }  
}
```

Output:

```
"info.properties"  
"info.properties".rtf  
.DS_Store  
.localized  
Alok news  
apache-tomcat-9.0.0.M19  
apache-tomcat-9.0.0.M19.tar  
bestreturn_org.rtf  
BIO DATA.pages  
BIO DATA.pdf  
BIO DATA.png  
struts2jars.zip  
workspace
```



Java File Example 4

```
import java.io.*;  
  
public class FileExample {  
    public static void main(String[] args) {  
        File dir=new File("/Users/sonoojaishwal/Documents");  
        File files[]=dir.listFiles();  
        for(File file:files){  
            System.out.println(file.getName()+" Can Write: "+file.canWrite()+"  
            Is Hidden: "+file.isHidden()+" Length: "+file.length()+" bytes");  
        }  
    }  
}
```

Output:

```
"info.properties" Can Write: true Is Hidden: false Length: 15 bytes  
"info.properties".rtf Can Write: true Is Hidden: false Length: 385 bytes  
.DS_Store Can Write: true Is Hidden: true Length: 36868 bytes  
.localized Can Write: true Is Hidden: true Length: 0 bytes  
Alok news Can Write: true Is Hidden: false Length: 850 bytes  
apache-tomcat-9.0.0.M19 Can Write: true Is Hidden: false Length: 476 bytes  
apache-tomcat-9.0.0.M19.tar Can Write: true Is Hidden: false Length: 13711360 bytes  
bestreturn_org.rtf Can Write: true Is Hidden: false Length: 389 bytes  
BIODATA.pages Can Write: true Is Hidden: false Length: 707985 bytes  
BIODATA.pdf Can Write: true Is Hidden: false Length: 69681 bytes  
BIODATA.png Can Write: true Is Hidden: false Length: 282125 bytes  
workspace Can Write: true Is Hidden: false Length: 1972 bytes
```

← Prev

Next →

Java FileDescriptor

FileDescriptor class serves as an handle to the underlying machine-specific structure representing an open file, an open **socket**, or another source or sink of bytes. The handle can be err, in or out.

The FileDescriptor class is used to create a **FileInputStream** or **FileOutputStream** to contain it.

Field

Modifier	Type	Field	Description
static	FileDescriptor	err	A handle to the standard error stream.
static	FileDescriptor	in	A handle to the standard input stream.
static	FileDescriptor	out	A handle to the standard output stream.

Constructors

Constructor	Description
FileDescriptor()	Constructs an (invalid) FileDescriptor object.

Method

Modifier and Type	Method	Description
void	sync()	It force all system buffers to synchronize with the underlying device.
boolean	valid()	It tests if this file descriptor object is valid.

Java FileDescriptor Example

```
import java.io.*;
public class FileDescriptorExample {
    public static void main(String[] args) {
        FileDescriptor fd = null;
```

```
byte[] b = { 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58 };

try {
    FileOutputStream fos = new FileOutputStream("Record.txt");
    FileInputStream fis = new FileInputStream("Record.txt");
    fd = fos.getFD();
    fos.write(b);
    fos.flush();
    fd.sync(); // confirms data to be written to the disk
    int value = 0;
    // for every available bytes
    while ((value = fis.read()) != -1) {
        char c = (char) value; // converts bytes to char
        System.out.print(c);
    }
    System.out.println("\nSync() successfully executed!!");
} catch (Exception e) {
    e.printStackTrace();
}
}
```

Output:

```
0123456789:  
Sync() successfully executed!!
```

Record.txt:

```
0123456789:
```

← Prev

Next →

Java - RandomAccessFile

This **class** is used for reading and writing to random access file. A random access file behaves like a large **array** of bytes. There is a cursor implied to the array called file **pointer**, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is **thrown**. It is a type of IOException.

Constructor

Constructor	Description
RandomAccessFile(File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Method

Modifier and Type	Method	Method
void	close()	It closes this random access file stream and releases any system resources associated with the stream.
FileChannel	getChannel()	It returns the unique FileChannel object associated with this file.
int	readInt()	It reads a signed 32-bit integer from this file.
String	readUTF()	It reads in a string from this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	writeDouble(double v)	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

void	writeFloat(float v)	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void	write(int b)	It writes the specified byte to this file.
int	read()	It reads a byte of data from this file.
long	length()	It returns the length of this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

Example

```

import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {
    static final String FILEPATH = "myFile.TXT";
    public static void main(String[] args) {
        try {
            System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
            writeToFile(FILEPATH, "I love my country and my people", 31);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static byte[] readFromFile(String filePath, int position, int size)
        throws IOException {
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(position);
        byte[] bytes = new byte[size];
        file.read(bytes);
        file.close();
        return bytes;
    }
}

```

```
private static void writeToFile(String filePath, String data, int position)
    throws IOException {
    RandomAccessFile file = new RandomAccessFile(filePath, "rw");
    file.seek(position);
    file.write(data.getBytes());
    file.close();
}
```

The myFile.TXT contains text "This class is used for reading and writing to random access file."

after running the program it will contains

This class is used for reading I love my country and my peoplele.

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java Scanner

Scanner class in Java is found in the `java.util` package. Java provides various ways to read input from the keyboard, the `java.util.Scanner` class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as `int`, `long`, `double`, `byte`, `float`, `short`, etc.

The Java Scanner class extends `Object` class and implements `Iterator` and `Closeable` interfaces.

The Java Scanner class provides `nextXXX()` methods to return the type of value such as `nextInt()`, `nextByte()`, `nextShort()`, `next()`, `nextLine()`, `nextDouble()`, `nextFloat()`, `nextBoolean()`, etc. To get a single character from the scanner, you can call `next().charAt(0)` method which returns a single character.

Java Scanner Class Declaration

```
public final class Scanner  
    extends Object  
    implements Iterator<String>
```

How to get Java Scanner

To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (`System.in`) in the constructor of `Scanner` class. For Example:

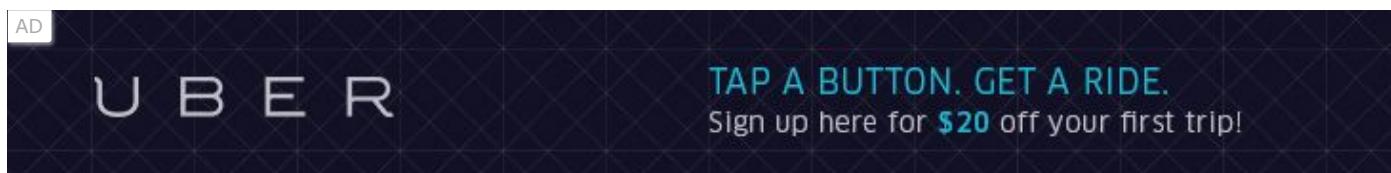
```
Scanner in = new Scanner(System.in);
```

To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of `Scanner` class. For Example:

```
Scanner in = new Scanner("Hello Javatpoint");
```

Java Scanner Class Constructors

SN	Constructor	Description
1)	Scanner(File source)	It constructs a new Scanner that produces values scanned from the specified file.
2)	Scanner(File source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.
3)	Scanner(InputStream source)	It constructs a new Scanner that produces values scanned from the specified input stream.
4)	Scanner(InputStream source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified input stream.
5)	Scanner(Readable source)	It constructs a new Scanner that produces values scanned from the specified source.
6)	Scanner(String source)	It constructs a new Scanner that produces values scanned from the specified string.
7)	Scanner(ReadableByteChannel source)	It constructs a new Scanner that produces values scanned from the specified channel.
8)	Scanner(ReadableByteChannel source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified channel.
9)	Scanner(Path source)	It constructs a new Scanner that produces values scanned from the specified file.
10)	Scanner(Path source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.



Java Scanner Class Methods

The following are the list of Scanner methods:

SN	Modifier & Type	Method	Description
1)	void	<code>close()</code>	It is used to close this scanner.
2)	pattern	<code>delimiter()</code>	It is used to get the Pattern which the Scanner class is currently using to match delimiters.
3)	Stream<MatchResult>	<code>findAll()</code>	It is used to find a stream of match results that match the provided pattern string.
4)	String	<code>findInLine()</code>	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
5)	string	<code>findWithinHorizon()</code>	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
6)	boolean	<code>hasNext()</code>	It returns true if this scanner has another token in its input.
7)	boolean	<code>hasNextBigDecimal()</code>	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the <code>nextBigDecimal()</code> method or not.
8)	boolean	<code>hasNextBigInteger()</code>	It is used to check if the next token in this scanner's input can be interpreted as a BigInteger using the <code>nextBigDecimal()</code> method or not.
9)	boolean	<code>hasNextBoolean()</code>	It is used to check if the next token in this scanner's input can be interpreted as a Boolean using the <code>nextBoolean()</code> method or not.
10)	boolean	<code>hasNextByte()</code>	It is used to check if the next token in this scanner's input can be interpreted as a Byte using the <code>nextBigDecimal()</code> method or not.

11)	boolean	<code>hasNextDouble()</code>	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextByte() method or not.
12)	boolean	<code>hasNextFloat()</code>	It is used to check if the next token in this scanner's input can be interpreted as a Float using the nextFloat() method or not.
13)	boolean	<code>hasNextInt()</code>	It is used to check if the next token in this scanner's input can be interpreted as an int using the nextInt() method or not.
14)	boolean	<code>hasNextLine()</code>	It is used to check if there is another line in the input of this scanner or not.
15)	boolean	<code>hasNextLong()</code>	It is used to check if the next token in this scanner's input can be interpreted as a Long using the nextLong() method or not.
16)	boolean	<code>hasNextShort()</code>	It is used to check if the next token in this scanner's input can be interpreted as a Short using the nextShort() method or not.
17)	IOException	<code>ioException()</code>	It is used to get the IOException last thrown by this Scanner's readable.
18)	Locale	<code>locale()</code>	It is used to get a Locale of the Scanner class.
19)	MatchResult	<code>match()</code>	It is used to get the match result of the last scanning operation performed by this scanner.
20)	String	<code>next()</code>	It is used to get the next complete token from the scanner which is in use.
21)	BigDecimal	<code>nextBigDecimal()</code>	It scans the next token of the input as a BigDecimal.
22)	BigInteger	<code>nextBigInteger()</code>	It scans the next token of the input as a BigInteger.

23)	boolean	<code>nextBoolean()</code>	It scans the next token of the input into a boolean value and returns that value.
24)	byte	<code>nextByte()</code>	It scans the next token of the input as a byte.
25)	double	<code>nextDouble()</code>	It scans the next token of the input as a double.
26)	float	<code>nextFloat()</code>	It scans the next token of the input as a float.
27)	int	<code>nextInt()</code>	It scans the next token of the input as an Int.
28)	String	<code>nextLine()</code>	It is used to get the input string that was skipped of the Scanner object.
29)	long	<code>nextLong()</code>	It scans the next token of the input as a long.
30)	short	<code>nextShort()</code>	It scans the next token of the input as a short.
31)	int	<code>radix()</code>	It is used to get the default radix of the Scanner use.
32)	void	<code>remove()</code>	It is used when remove operation is not supported by this implementation of Iterator.
33)	Scanner	<code>reset()</code>	It is used to reset the Scanner which is in use.
34)	Scanner	<code>skip()</code>	It skips input that matches the specified pattern, ignoring delimiters
35)	Stream<String>	<code>tokens()</code>	It is used to get a stream of delimiter-separated tokens from the Scanner object which is in use.
36)	String	<code>toString()</code>	It is used to get the string representation of Scanner using.

37)	Scanner	<code>useDelimiter()</code>	It is used to set the delimiting pattern of the Scanner which is in use to the specified pattern.
38)	Scanner	<code>useLocale()</code>	It is used to sets this scanner's locale object to the specified locale.
39)	Scanner	<code>useRadix()</code>	It is used to set the default radix of the Scanner which is in use to the specified radix.

Example 1

Let's see a simple example of Java Scanner where we are getting a single input from the user. Here, we are asking for a string through `in.nextLine()` method.

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
    Scanner in = new Scanner(System.in);
    System.out.print("Enter your name: ");
    String name = in.nextLine();
    System.out.println("Name is: " + name);
    in.close();
}
}
```

Output:

```
Enter your name: sonoo jaiswal
Name is: sonoo jaiswal
```

Example 2

```
import java.util.*;
public class ScannerClassExample1 {
public static void main(String args[]){
    String s = "Hello, This is JavaTpoint.";
}
```

```
//Create scanner Object and pass string in it
Scanner scan = new Scanner(s);
//Check if the scanner has a token
System.out.println("Boolean Result: " + scan.hasNext());
//Print the string
System.out.println("String: " + scan.nextLine());
scan.close();
System.out.println("-----Enter Your Details-----");
Scanner in = new Scanner(System.in);
System.out.print("Enter your name: ");
String name = in.next();
System.out.println("Name: " + name);
System.out.print("Enter your age: ");
int i = in.nextInt();
System.out.println("Age: " + i);
System.out.print("Enter your salary: ");
double d = in.nextDouble();
System.out.println("Salary: " + d);
in.close();
}
}
```

Output:

```
Boolean Result: true
String: Hello, This is JavaTpoint.
-----
Enter Your Details-----
Enter your name: Abhishek
Name: Abhishek
Enter your age: 23
Age: 23
Enter your salary: 25000
Salary: 25000.0
```

Example 3

```
import java.util.*;
```

```
public class ScannerClassExample2 {  
    public static void main(String args[]){  
        String str = "Hello/This is JavaTpoint/My name is Abhishek.";  
        //Create scanner with the specified String Object  
        Scanner scanner = new Scanner(str);  
        System.out.println("Boolean Result: "+scanner.hasNextBoolean());  
        //Change the delimiter of this scanner  
        scanner.useDelimiter("/");  
        //Printing the tokenized Strings  
        System.out.println("---Tokenizes String---");  
        while(scanner.hasNext()){  
            System.out.println(scanner.next());  
        }  
        //Display the new delimiter  
        System.out.println("Delimiter used: " +scanner.delimiter());  
        scanner.close();  
    }  
}
```

Output:

```
Boolean Result: false  
---Tokenizes String---  
Hello  
This is JavaTpoint  
My name is Abhishek.  
Delimiter used: /
```

← Prev

Next →

java.io.PrintStream class

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

Commonly used methods of PrintStream class

There are many methods in PrintStream class. Let's see commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.
- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line.
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.
- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

Example of java.io.PrintStream class

In this example, we are simply printing integer and string values.

```
import java.io.*;
class PrintStreamTest{
    public static void main(String args[])throws Exception{
        FileOutputStream fout=new FileOutputStream("mfile.txt");
        PrintStream pout=new PrintStream(fout);
        pout.println(1900);
        pout.println("Hello Java");
        pout.println("Welcome to Java");
        pout.close();
        fout.close();
    }
}
```

[download this PrintStream example](#)

Example of printf() method of java.io.PrintStream class:

Let's see the simple example of printing integer value by format specifier.

```
class PrintStreamTest{
    public static void main(String args[]){
        int a=10;
        System.out.printf("%d",a); //Note, out is the object of PrintStream class
    }
}
```

Output:10

Compressing and Decompressing File

The DeflaterOutputStream and InflaterInputStream classes provide mechanism to compress and decompress the data in the **deflate compression format**.

DeflaterOutputStream class

The DeflaterOutputStream class is used to compress the data in the deflate compression format. It provides facility to the other compression filters, such as GZIPOutputStream.

Example of Compressing file using DeflaterOutputStream class

In this example, we are reading data of a file and compressing it into another file using DeflaterOutputStream class. You can compress any file, here we are compressing the Deflater.java file

```
import java.io.*;
import java.util.zip.*;
class Compress{
public static void main(String args[]){
try{
FileInputStream fin=new FileInputStream("Deflater.java");
FileOutputStream fout=new FileOutputStream("def.txt");
DeflaterOutputStream out=new DeflaterOutputStream(fout);

int i;
while((i=fin.read())!=-1){
out.write((byte)i);
out.flush();
}
fin.close();
out.close();
}catch(Exception e){System.out.println(e);}
System.out.println("rest of the code");
}
}
```

[download this example](#)

InflaterInputStream class

The InflaterInputStream class is used to decompress the file in the deflate compression format. It provides facility to the other decompression filters, such as GZIPInputStream class.

Example of decompressing file using InflaterInputStream class

In this example, we are decompressing the compressed file def.txt into D.java .

```
import java.io.*;
import java.util.zip.*;
class DeCompress{
public static void main(String args[]){
try{
FileInputStream fin=new FileInputStream("def.txt");
InflaterInputStream in=new InflaterInputStream(fin);
FileOutputStream fout=new FileOutputStream("D.java");

int i;
while((i=in.read())!=-1){
fout.write((byte)i);
fout.flush();
}
fin.close();
fout.close();
in.close();
}catch(Exception e){System.out.println(e);}
System.out.println("rest of the code");
}
}
```

[download this example](#)

PipedInputStream and PipedOutputStream classes

The PipedInputStream and PipedOutputStream classes can be used to read and write data simultaneously. Both streams are connected with each other using the connect() method of the PipedOutputStream class.

Example of PipedInputStream and PipedOutputStream classes using threads

Here, we have created two threads t1 and t2. The **t1** thread writes the data using the PipedOutputStream object and the **t2** thread reads the data from that pipe using the PipedInputStream object. Both the piped stream object are connected with each other.

```
import java.io.*;
class PipedWR{
public static void main(String args[])throws Exception{
final PipedOutputStream pout=new PipedOutputStream();
final PipedInputStream pin=new PipedInputStream();

pout.connect(pin); //connecting the streams
//creating one thread t1 which writes the data
Thread t1=new Thread(){
public void run(){
for(int i=65;i<=90;i++){
try{
pout.write(i);
Thread.sleep(1000);
}catch(Exception e){}
}
}
};

//creating another thread t2 which reads the data
Thread t2=new Thread(){
public void run(){
try{
for(int i=65;i<=90;i++)
System.out.println(pin.read());
}
}
};
```

```
}catch(Exception e){}
}
};

//starting both threads
t1.start();
t2.start();
}}
```

[download this example](#)

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

Java I/O Tutorial

Java I/O (Input and Output) is used *to process the input and produce the output.*

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and an error** message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Let's see the code to get **input** from console.

```
int i=System.in.read();//returns ASCII code of 1st character
System.out.println((char)i);//will print the character
```

Do You Know?

- How to write a common data to multiple files using a single stream only?
- How can we access multiple files by a single stream?
- How can we improve the performance of Input and Output operation?
- How many ways can we read data from the keyboard?
- What does the console class?

- o How to compress and uncompress the data of a file?

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

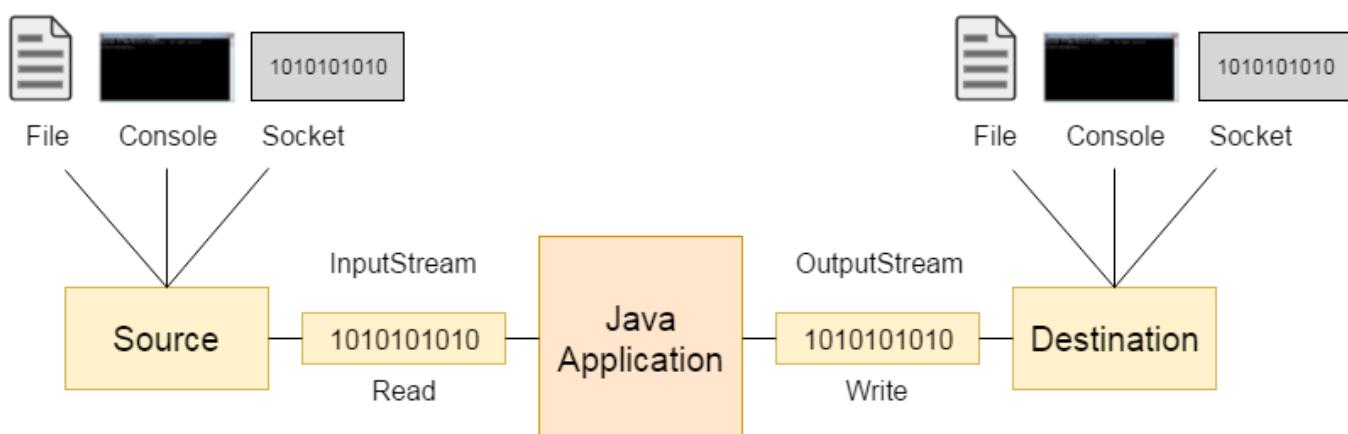
OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



OutputStream class

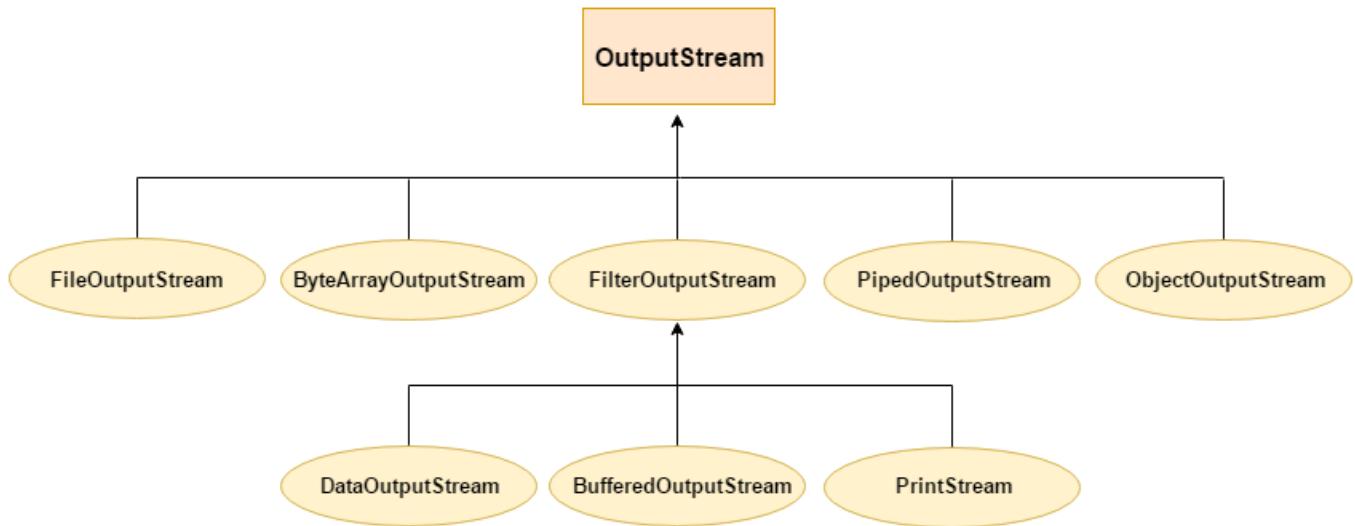
OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int) throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[]) throws IOException	is used to write an array of byte to the current output stream.

3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



AD



VIDEO STREAMING FOR THE AWAKENED MIND

[GET STARTED NOW](#)

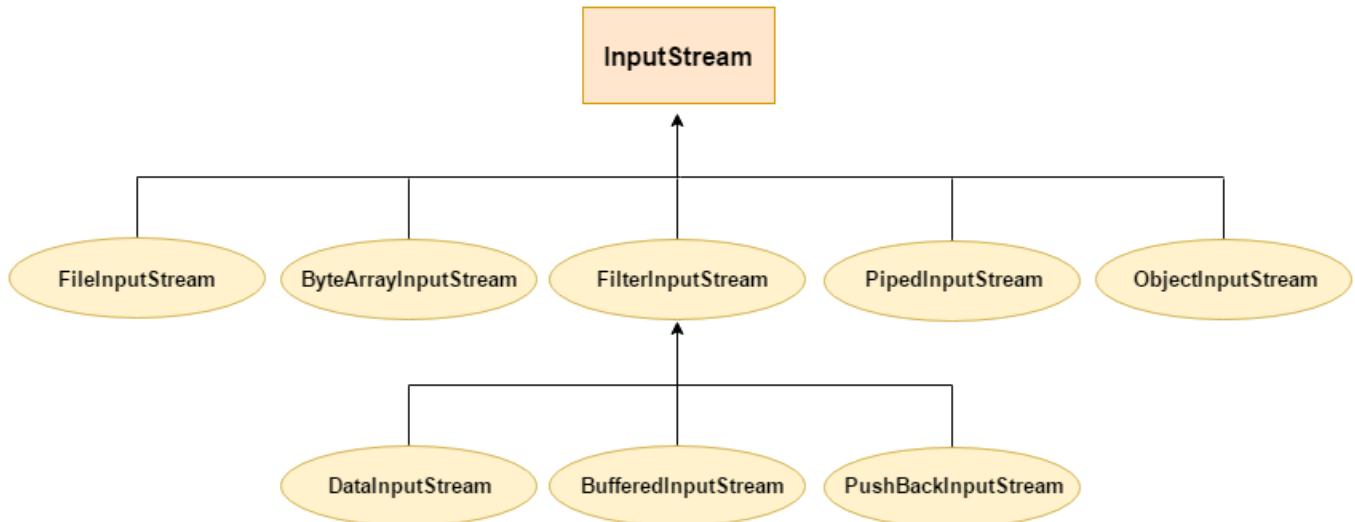
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



← Prev

Next →

AD

[For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a [file](#).

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use [FileWriter](#) than FileOutputStream.

OutputStream class declaration

Let's see the declaration for Java.io.FileOutputStream class:

```
public class FileOutputStream extends OutputStream
```

OutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write len bytes from the byte array starting at offset off to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

Java FileOutputStream Example 1: write byte

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
```

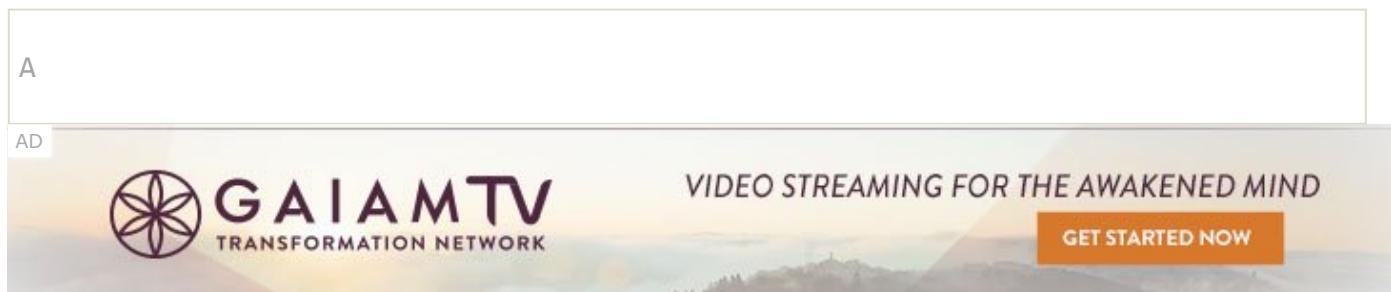
```
public static void main(String args[]){  
    try{  
        FileOutputStream fout=new FileOutputStream("D:\\testout.txt");  
        fout.write(65);  
        fout.close();  
        System.out.println("success...");  
    }catch(Exception e){System.out.println(e);}  
}
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **A**.

testout.txt



Java FileOutputStream example 2: write string

```
import java.io.FileOutputStream;  
public class FileOutputStreamExample {  
    public static void main(String args[]){  
        try{  
            FileOutputStream fout=new FileOutputStream("D:\\testout.txt");  
            String s="Welcome to javaTpoint.";  
            byte b[]={s.getBytes()}; //converting string into byte array  
            fout.write(b);  
            fout.close();  
            System.out.println("success...");  
        }catch(Exception e){System.out.println(e);}  
    }  
}
```

```
}
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **Welcome to javaTpoint.**

testout.txt

```
Welcome to javaTpoint.
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Java FileInputStream Class

Java FileInputStream class obtains input bytes from a [file](#). It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use [FileReader](#) class.

Java FileInputStream class declaration

Let's see the declaration for java.io.FileInputStream class:

```
public class FileInputStream extends InputStream
```

Java FileInputStream class methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to b.length bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to len bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream .

Java FileInputStream example 1: read single character

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("D:\\testout.txt");
            int i=fin.read();
            System.out.print((char)i);

            fin.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

Note: Before running the code, a text file named as "**testout.txt**" is required to be created. In this file, we are having following content:

```
Welcome to javatpoint.
```

After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.

Output:

```
W
```

AD

Java FileInputStream example 2: read all characters

```
package com.javatpoint;

import java.io.FileInputStream;
public class DataStreamExample {
```

```
public static void main(String args[]){
    try{
        FileInputStream fin=new FileInputStream("D:\\testout.txt");
        int i=0;
        while((i=fin.read())!=-1){
            System.out.print((char)i);
        }
        fin.close();
    }catch(Exception e){System.out.println(e);}
}
```

Output:

```
Welcome to javaTpoint
```

← Prev

Next →

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java BufferedOutputStream Class

Java BufferedOutputStream **class** is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

```
OutputStream os= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt")
```

Java BufferedOutputStream class declaration

Let's see the declaration for Java.io.BufferedOutputStream class:

```
public class BufferedOutputStream extends FilterOutputStream
```

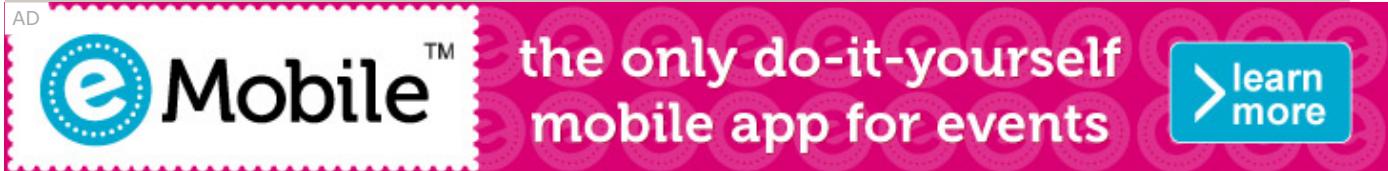
Java BufferedOutputStream class constructors

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Java BufferedOutputStream class methods

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.

void write(byte[] b, int off, int len)	It writes the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.



Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the [FileOutputStream object](#). The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
package com.javatpoint;
import java.io.*;
public class BufferedOutputStreamExample{
public static void main(String args[])throws Exception{
    FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
    BufferedOutputStream bout=new BufferedOutputStream(fout);
    String s="Welcome to javaTpoint.";
    byte b[]={s.getBytes()};
    bout.write(b);
    bout.flush();
    bout.close();
    fout.close();
    System.out.println("success");
}
}
```

Output:

Success

testout.txt

Welcome to javaTpoint.

Java BufferedInputStream Class

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

Java BufferedInputStream class declaration

Let's see the declaration for Java.io.BufferedInputStream class:

```
public class BufferedInputStream extends FilterInputStream
```

Java BufferedInputStream class constructors

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves its argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves its argument, the input stream IS, for later use.

Java BufferedInputStream class methods

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It reads the next byte of data from the input stream.

int read(byte[] b, int off, int ln)	It reads the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

Example of Java BufferedInputStream

Let's see the simple example to read data of **file** using **BufferedInputStream**:

```
package com.javatpoint;

import java.io.*;
public class BufferedInputStreamExample{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("D:\\testout.txt");
            BufferedInputStream bin=new BufferedInputStream(fin);
            int i;
            while((i=bin.read())!=-1){
                System.out.print((char)i);
            }
            bin.close();
            fin.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

Here, we are assuming that you have following data in "**testout.txt**" file:

javatpoint

Output:

javatpoint

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

 [Splunk tutorial](#)

Splunk

 [SPSS tutorial](#)

SPSS

 [Swagger tutorial](#)

Swagger

 [T-SQL tutorial](#)

Transact-SQL

Java Networking

Java Networking is a concept of connecting two or more computing devices together so that we can share resources.

Java socket programming provides facility to share data between different computing devices.

Advantage of Java Networking

1. Sharing resources
2. Centralize software management

Do You Know ?

- How to perform connection-oriented Socket Programming in networking ?
- How to display the data of any online web page ?
- How to get the IP address of any host name e.g. www.google.com ?
- How to perform connection-less socket programming in networking ?

The java.net package supports two protocols,

1. **TCP:** Transmission Control Protocol provides reliable communication between the sender and receiver. TCP is used along with the Internet Protocol referred as TCP/IP.
2. **UDP:** User Datagram Protocol provides a connection-less protocol service by allowing packet of data to be transferred along two or more nodes

Java Networking Terminology

The widely used Java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket

1) IP Address

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

It is a logical address that can be changed.

2) Protocol

A protocol is a set of rules basically that is followed for communication. For example:

- TCP
- FTP
- Telnet
- SMTP
- POP etc.

3) Port Number

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

4) MAC Address

MAC (Media Access Control) address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC address.

For example, an ethernet card may have a **MAC** address of 00:0d:83::b1:c0:8e.

5) Connection-oriented and connection-less protocol

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.

6) Socket

A socket is an endpoint between two way communications.

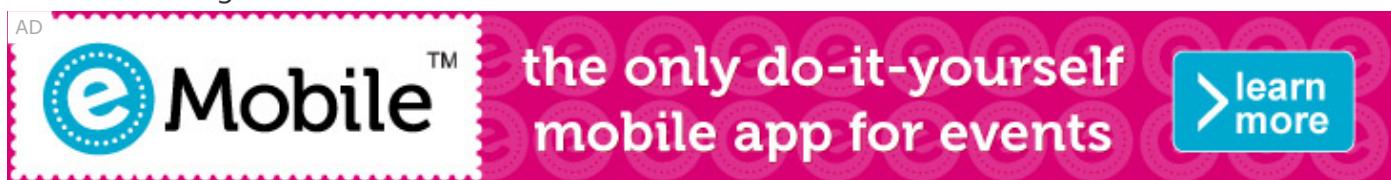
Visit next page for Java socket programming.

java.net package

The java.net package can be divided into two sections:

1. **A Low-Level API:** It deals with the abstractions of addresses i.e. networking identifiers, Sockets i.e. bidirectional data communication mechanism and Interfaces i.e. network interfaces.
2. **A High Level API:** It deals with the abstraction of URLs i.e. Universal Resource Identifier, URLs i.e. Universal Resource Locator, and Connections i.e. connections to the resource pointed by URLs.

The java.net package provides many classes to deal with networking applications in Java. A list of these classes is given below:



- Authenticator
- CacheRequest
- CacheResponse
- ContentHandler
- CookieHandler
- CookieManager
- DatagramPacket
- DatagramSocket
- DatagramSocketImpl
- InterfaceAddress
- JarURLConnection
- MulticastSocket
- InetSocketAddress
- InetAddress
- Inet4Address
- Inet6Address
- IDN
- HttpURLConnection

- HttpCookie
- NetPermission
- NetworkInterface
- PasswordAuthentication
- Proxy
- ProxySelector
- ResponseCache
- SecureCacheResponse
- ServerSocket
- Socket
- SocketAddress
- SocketImpl
- SocketPermission
- StandardSocketOptions
- URI
- URL
- URLClassLoader
- URLConnection
- URLDecoder
- URLEncoder
- URLStreamHandler



List of interfaces available in java.net package:

- ContentHandlerFactory
- CookiePolicy
- CookieStore
- DatagramSocketImplFactory
- FileNameMap

- SocketOption<T>
- SocketOptions
- SocketImplFactory
- URLStreamHandlerFactory
- ProtocolFamily

What we will learn in Networking Tutorial

- Networking and Networking Terminology
- Socket Programming (Connection-oriented)
- URL class
- Displaying data of a webpage by URLConnection class
- InetAddress class
- DatagramSocket and DatagramPacket (Connection-less)

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

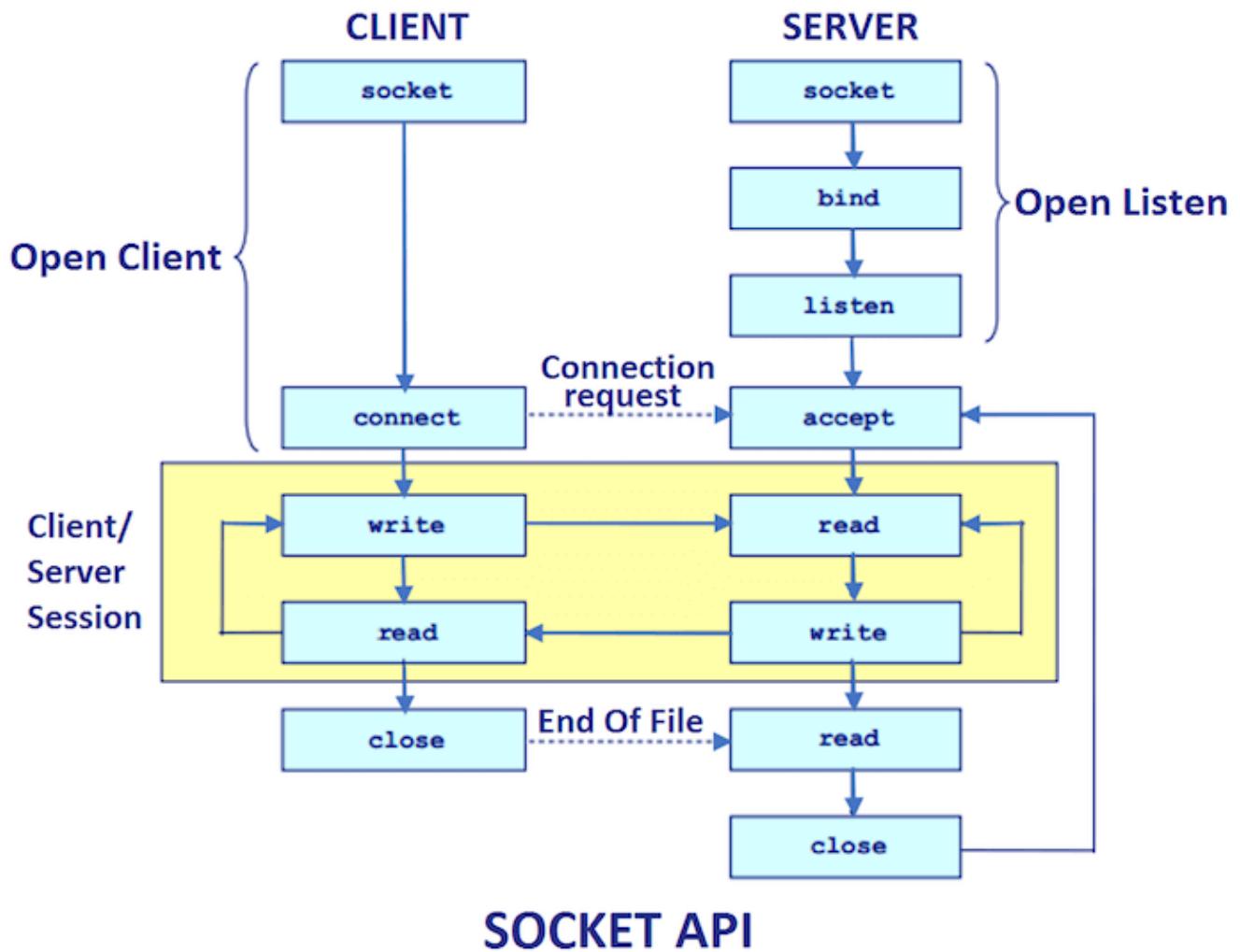
Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.



Socket class

A socket is simply an endpoint for communications between the machines. The `Socket` class can be used to create a socket.

Important methods

Method	Description
1) <code>public InputStream getInputStream()</code>	returns the <code>InputStream</code> attached with this socket.
2) <code>public OutputStream getOutputStream()</code>	returns the <code>OutputStream</code> attached with this socket.
3) <code>public synchronized void close()</code>	closes this socket

ServerSocket class

The `ServerSocket` class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

Creating Server:

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

```
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept(); //establishes connection and waits for the client
```

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

```
Socket s=new Socket("localhost",6666);
```

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

File: MyServer.java

```
import java.io.*;
import java.net.*;
public class MyServer {
    public static void main(String[] args){
```

```
try{
    ServerSocket ss=new ServerSocket(6666);
    Socket s=ss.accept();//establishes connection
    DataInputStream dis=new DataInputStream(s.getInputStream());
    String str=(String)dis.readUTF();
    System.out.println("message= "+str);
    ss.close();
}catch(Exception e){System.out.println(e);}
}
```

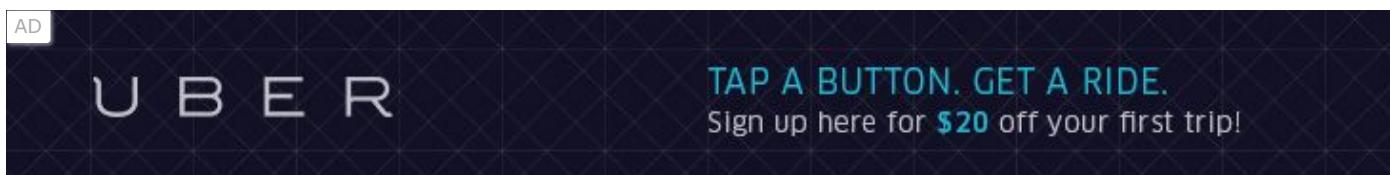
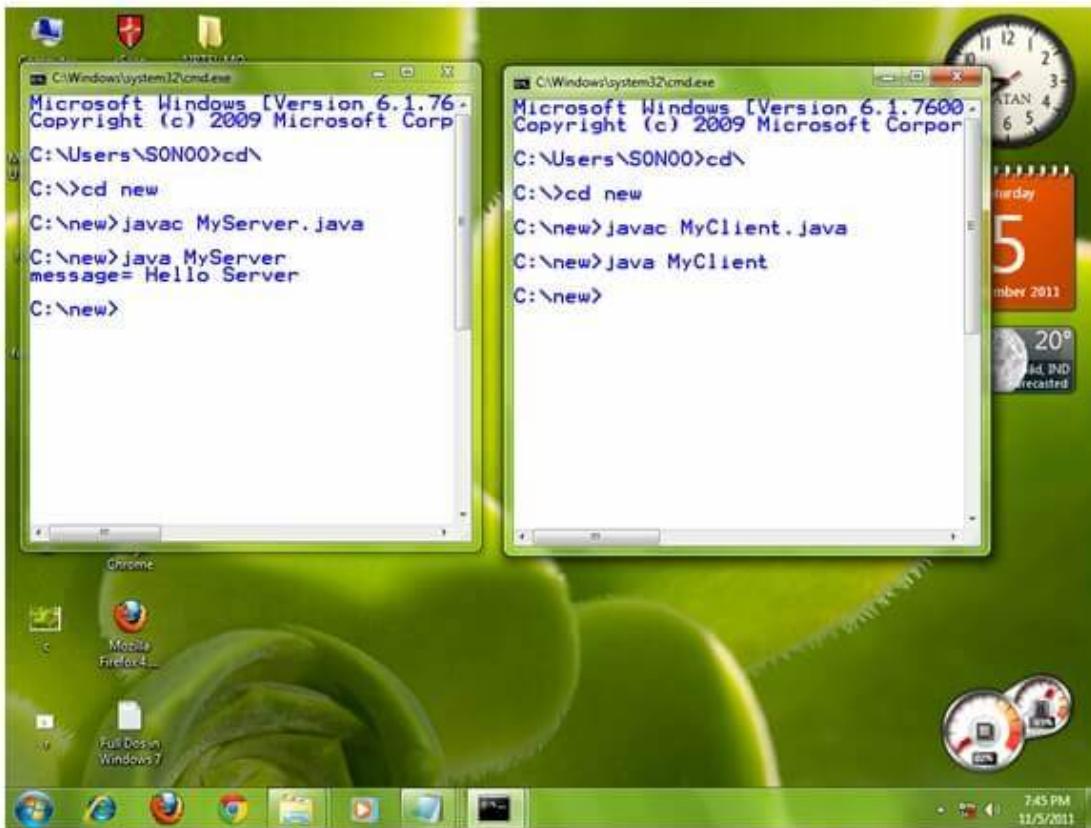
File: MyClient.java

```
import java.io.*;
import java.net.*;
public class MyClient {
    public static void main(String[] args) {
        try{
            Socket s=new Socket("localhost",6666);
            DataOutputStream dout=new DataOutputStream(s.getOutputStream());
            dout.writeUTF("Hello Server");
            dout.flush();
            dout.close();
            s.close();
        }catch(Exception e){System.out.println(e);}
    }
}
```

[download this example](#)

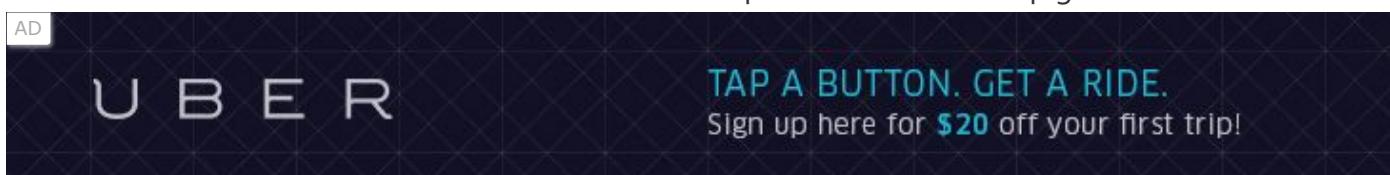
To execute this program open two command prompts and execute each program at each command prompt as displayed in the below figure.

After running the client application, a message will be displayed on the server console.



Example of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text. Then server will write to the client and client will receive and print the text. The step goes on.



File: MyServer.java

```
import java.net.*;
import java.io.*;
class MyServer{
public static void main(String args[])throws Exception{
    ServerSocket ss=new ServerSocket(3333);
    Socket s=ss.accept();
    DataInputStream din=new DataInputStream(s.getInputStream());
    DataOutputStream dout=new DataOutputStream(s.getOutputStream());
```

```
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```

```
String str="",str2="";
while(!str.equals("stop")){
    str=din.readUTF();
    System.out.println("client says: "+str);
    str2=br.readLine();
    dout.writeUTF(str2);
    dout.flush();
}
din.close();
s.close();
ss.close();
}}
```

File: MyClient.java

```
import java.net.*;
import java.io.*;
class MyClient{
public static void main(String args[])throws Exception{
    Socket s=new Socket("localhost",3333);
    DataInputStream din=new DataInputStream(s.getInputStream());
    DataOutputStream dout=new DataOutputStream(s.getOutputStream());
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

    String str="",str2="";
    while(!str.equals("stop")){
        str=br.readLine();
        dout.writeUTF(str);
        dout.flush();
        str2=din.readUTF();
        System.out.println("Server says: "+str2);
    }

    dout.close();
    s.close();
}}
```

}}

[← Prev](#)[Next →](#)

AD

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials



Splunk



SPSS

Swagger
tutorial

Swagger



Transact-SQL



Tumblr



ReactJS

Regex
tutorial

Regex

Reinforcement
learning
Learning

Java URL

The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. For example:

```
https://www.javatpoint.com/java-tutorial
```



A URL contains many information:

1. **Protocol:** In this case, http is the protocol.
2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.
3. **Port Number:** It is an optional attribute. If we write http://www.javatpoint.com:80/sonoojaiswal/, 80 is the port number. If port number is not mentioned in the URL, it returns -1.
4. **File Name or directory name:** In this case, index.jsp is the file name.

Constructors of Java URL class

URL(String spec)

Creates an instance of a URL from the String representation.

URL(String protocol, String host, int port, String file)

Creates an instance of a URL from the given protocol, host, port number, and file.

URL(String protocol, String host, int port, String file, URLStreamHandler handler)

Creates an instance of a URL from the given protocol, host, port number, file, and handler.

URL(String protocol, String host, String file)

Creates an instance of a URL from the given protocol name, host name, and file name.

URL(URL context, String spec)

Creates an instance of a URL by parsing the given spec within a specified context.

URL(URL context, String spec, URLStreamHandler handler)

Creates an instance of a URL by parsing the given spec with the specified handler within a given context.

Commonly used methods of Java URL class

The `java.net.URL` class provides many methods. The important methods of URL class are given below.

Method	Description
<code>public String getProtocol()</code>	it returns the protocol of the URL.
<code>public String getHost()</code>	it returns the host name of the URL.
<code>public String getPort()</code>	it returns the Port Number of the URL.
<code>public String getFile()</code>	it returns the file name of the URL.
<code>public String getAuthority()</code>	it returns the authority of the URL.
<code>public String toString()</code>	it returns the string representation of the URL.
<code>public String getQuery()</code>	it returns the query string of the URL.
<code>public String getDefaultPort()</code>	it returns the default port of the URL.
<code>public URLConnection openConnection()</code>	it returns the instance of <code>URLConnection</code> i.e. associated with this URL.
<code>public boolean equals(Object obj)</code>	it compares the URL with the given object.
<code>public Object getContent()</code>	it returns the content of the URL.
<code>public String getRef()</code>	it returns the anchor or reference of the URL.
<code>public URI toURI()</code>	it returns a <code>URI</code> of the URL.

Example of Java URL class

```
//URLDemo.java
import java.net.*;
public class URLEDemo{
    public static void main(String[] args){
        try{

```

```

URL url=new URL("http://www.javatpoint.com/java-tutorial");

System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("File Name: "+url.getFile());

}catch(Exception e){System.out.println(e);}
}
}

```

Test it Now

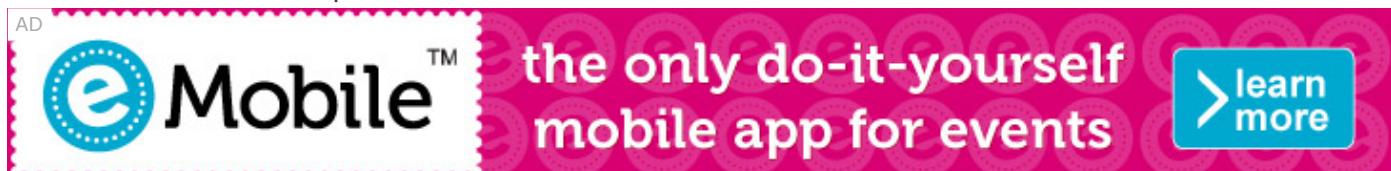
Output:

```

Protocol: http
Host Name: www.javatpoint.com
Port Number: -1
File Name: /java-tutorial

```

Let us see another example URL class in Java.



```

//URLDemo.java
import java.net.*;
public class URLEDemo{
public static void main(String[] args){
try{
URL url=new URL("https://www.google.com/search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8");
System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
System.out.println("Port Number: "+url.getPort());
System.out.println("Default Port Number: "+url.getDefaultPort());
System.out.println("Query String: "+url.getQuery());
}
}

```

```
System.out.println("Path: "+url.getPath());
System.out.println("File: "+url.getFile());

}catch(Exception e){System.out.println(e);}

}
```

Output:

```
Protocol: https
Host Name: www.google.com
Port Number: -1
Default Port Number: 443
Query String: q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
Path: /search
File: /search?q=javatpoint&oq=javatpoint&sourceid=chrome&ie=UTF-8
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

Java URLConnection Class

The **Java URLConnection** class represents a communication link between the URL and the application. It can be used to read and write data to the specified resource referred by the URL.

What is the URL?

- URL is an abbreviation for Uniform Resource Locator. An URL is a form of string that helps to find a resource on the World Wide Web (WWW).
- URL has two components:
 1. The protocol required to access the resource.
 2. The location of the resource.

Features of URLConnection class

1. URLConnection is an abstract class. The two subclasses HttpURLConnection and JarURLConnection makes the connection between the client Java program and URL resource on the internet.
2. With the help of URLConnection class, a user can read and write to and from any resource referenced by an URL object.
3. Once a connection is established and the Java program has an URLConnection object, we can use it to read or write or get further information like content length, etc.

Constructors

Constructor	Description
1) protected URLConnection(URL url)	It constructs a URL connection to the specified URL.

URLConnection Class Methods

Method	Description
void addRequestProperty(String key, String value)	It adds a general request property specified by a key-value pair

void connect()	It opens a communications link to the resource referenced by this URL, if such a connection has not already been established.
boolean getAllowUserInteraction()	It returns the value of the allowUserInteraction field for the object.
int getConnectionTimeout()	It returns setting for connect timeout.
Object getContent()	It retrieves the contents of the URL connection.
Object getContent(Class[] classes)	It retrieves the contents of the URL connection.
String getContentEncoding()	It returns the value of the content-encoding header field.
int getContentLength()	It returns the value of the content-length header field.
long getContentLengthLong()	It returns the value of the content-length header field as long.
String getContentType()	It returns the value of the date header field.
long getDate()	It returns the value of the date header field.
static boolean getDefaultAllowUserInteraction()	It returns the default value of the allowUserInteraction field.
boolean getDefaultUseCaches()	It returns the default value of an URLConnnection's useCaches flag.
boolean getDoInput()	It returns the value of the URLConnection's doInput flag.
boolean getDoOutput()	It returns the value of the URLConnection's doOutput flag.
long getExpiration()	It returns the value of the expires header files.

static FileNameMap getFilenameMap()	It loads the filename map from a data file.
String getHeaderField(int n)	It returns the value of n th header field
String getHeaderField(String name)	It returns the value of the named header field.
long getHeaderFieldDate(String name, long Default)	It returns the value of the named field parsed as a number.
int getHeaderFieldInt(String name, int Default)	It returns the value of the named field parsed as a number.
String getHeaderFieldKey(int n)	It returns the key for the n th header field.
long getHeaderFieldLong(String name, long Default)	It returns the value of the named field parsed as a number.
Map<String, List<String>> getHeaderFields()	It returns the unmodifiable Map of the header field.
long getLastModified()	It returns the value of the object's ifModifiedSince field.
InputStream getInputStream()	It returns an input stream that reads from the open condition.
OutputStream getOutputStream()	It returns an output stream that writes to the connection.
Permission getPermission()	It returns a permission object representing the permission necessary to make the connection represented by the object.
int getReadTimeout()	It returns setting for read timeout.

Map<String, List<String>> getRequestProperties()	It returns the value of the named general request property for the connection.
URL getURL()	It returns the value of the URLConnection's URL field.
boolean getUseCaches()	It returns the value of the URLConnection's useCaches field.
Static String guessContentTypeFromName(String fname)	It tries to determine the content type of an object, based on the specified file component of a URL.
static String guessContentTypeFromStream(InputStream is)	It tries to determine the type of an input stream based on the characters at the beginning of the input stream.
void setAllowUserInteraction(boolean allowuserinteraction)	It sets the value of the allowUserInteraction field of this URLConnection.
static void setContentHandlerFactory(ContentHandlerFactory fac)	It sets the ContentHandlerFactory of an application.
static void setDefaultAllowUserInteraction(boolean defaultallowuserinteraction)	It sets the default value of the allowUserInteraction field for all future URLConnection objects to the specified value.
void steDafaultUseCaches(boolean defaultusecaches)	It sets the default value of the useCaches field to the specified value.
void setDoInput(boolean doinput)	It sets the value of the doInput field for this URLConnection to the specified value.
void setDoOutput(boolean dooutput)	It sets the value of the doOutput field for the URLConnection to the specified value.

How to get the object of URLConnection Class

The openConnection() method of the URL class returns the object of URLConnection class.

Syntax:

```
public URLConnection openConnection()throws IOException{}
```

Displaying Source Code of a Webpage by URLConnecton Class

The URLConnection class provides many methods. We can display all the data of a webpage by using the getInputStream() method. It returns all the data of the specified URL in the stream that can be read and displayed.

Example of Java URLConnection Class

```
import java.io.*;
import java.net.*;
public class URLConnectionExample {
    public static void main(String[] args){
        try{
            URL url=new URL("http://www.javatpoint.com/java-tutorial");
            URLConnection urlcon=url.openConnection();
            InputStream stream=urlcon.getInputStream();
            int i;
            while((i=stream.read())!=-1){
                System.out.print((char)i);
            }
        }catch(Exception e){System.out.println(e);}
    }
}
```

Java HttpURLConnection class

The **Java HttpURLConnection** class is http specific URLConnection. It works for HTTP protocol only.

By the help of HttpURLConnection class, you can retrieve information of any HTTP URL such as header information, status code, response code etc.

The `java.net.HttpURLConnection` is subclass of `URLConnection` class.

HttpURLConnection Class Constructor

Constructor	Description
<code>protected HttpURLConnection(URL u)</code>	It constructs the instance of <code>HttpURLConnection</code> class.

Java HttpURLConnection Methods

Method	Description
<code>void disconnect()</code>	It shows that other requests from the server are unlikely in the near future.
<code>InputStream getErrorStream()</code>	It returns the error stream if the connection failed but the server sent useful data.
<code>Static boolean getFollowRedirects()</code>	It returns a boolean value to check whether or not HTTP redirects should be automatically followed.
<code>String getHeaderField(int n)</code>	It returns the value of nth header file.
<code>long getHeaderFieldDate(String name, long Default)</code>	It returns the value of the named field parsed as a date.
<code>String getHeaderFieldKey(int n)</code>	It returns the key for the nth header file.
<code>boolean getInstanceFollowRedirects()</code>	It returns the value of <code>HttpURLConnection</code> 's instance <code>FollowRedirects</code> field.
<code>Permission getPermission()</code>	It returns the <code>SocketPermission</code> object representing the permission to connect to the destination host and port.

<code>String getRequestMethod()</code>	It gets the request method.
<code>int getResponseCode()</code>	It gets the response code from an HTTP response message.
<code>String getResponseMessage()</code>	It gets the response message sent along with the response code from a server.
<code>void setChunkedStreamingMode(int chunklen)</code>	The method is used to enable streaming of a HTTP request body without internal buffering, when the content length is not known in advance.
<code>void setFixedLengthStreamingMode(int contentlength)</code>	The method is used to enable streaming of a HTTP request body without internal buffering, when the content length is known in advance.
<code>void setFixedLengthStreamingMode(long contentlength)</code>	The method is used to enable streaming of a HTTP request body without internal buffering, when the content length is not known in advance.
<code>static void setFollowRedirects(boolean set)</code>	It sets whether HTTP redirects (requests with response code) should be automatically followed by HttpURLConnection class.
<code>void setInstanceFollowRedirects(boolean followRedirects)</code>	It sets whether HTTP redirects (requests with response code) should be automatically followed by instance of HttpURLConnection class.
<code>void setRequestMethod(String method)</code>	Sets the method for the URL request, one of: GET POST HEAD OPTIONS PUT DELETE TRACE are legal, subject to protocol restrictions.
<code>abstract boolean usingProxy()</code>	It shows if the connection is going through a proxy.

How to get the object of HttpURLConnection class

The `openConnection()` method of URL class returns the object of URLConnection class.

Syntax:

```
public URLConnection openConnection()throws IOException{}
```

You can typecast it to HttpURLConnection type as given below.

```
URL url=new URL("http://www.javatpoint.com/java-tutorial");
HttpURLConnection huc=(HttpURLConnection)url.openConnection();
```

AD



VIDEO STREAMING FOR THE AWAKENED MIND

GET STARTED NOW

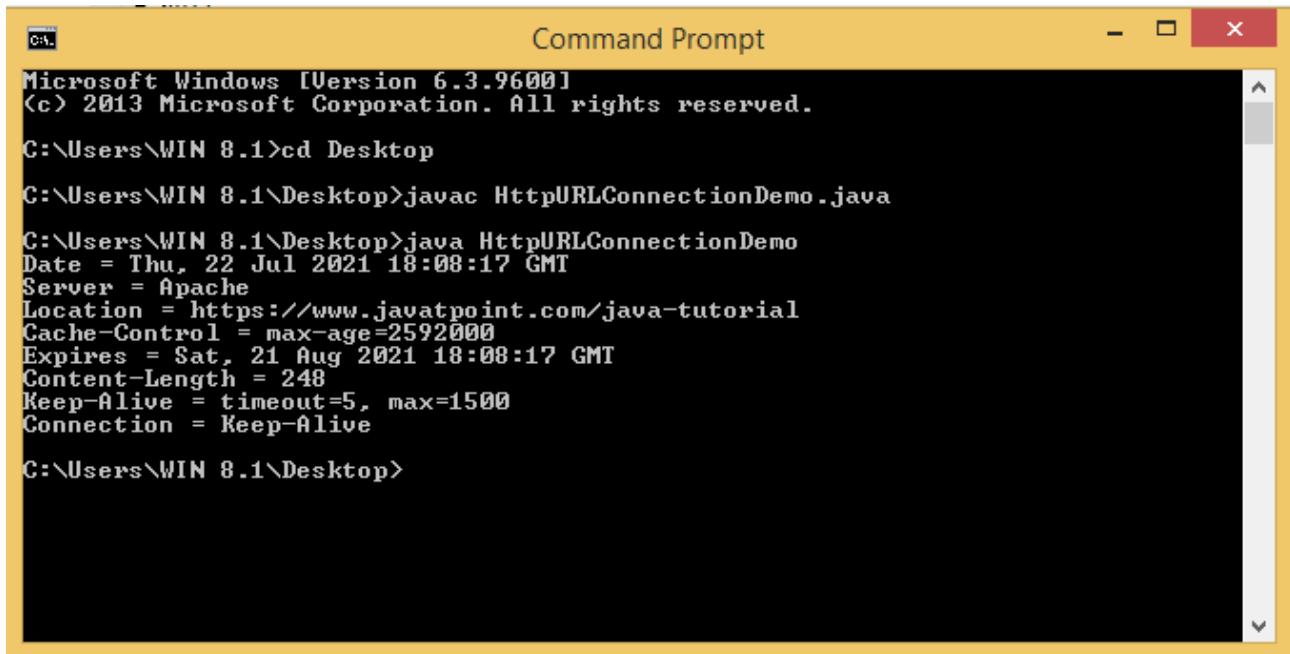
Java HttpURLConnection Example

```
import java.io.*;
import java.net.*;
public class HttpURLConnectionDemo{
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
HttpURLConnection huc=(HttpURLConnection)url.openConnection();
for(int i=1;i<=8;i++){
System.out.println(huc.getHeaderFieldKey(i)+" = "+huc.getHeaderField(i));
}
huc.disconnect();
}catch(Exception e){System.out.println(e);}
}
}
```

Test it Now

Output:

```
Date = Thu, 22 Jul 2021 18:08:17 GMT
Server = Apache
Location = https://www.javatpoint.com/java-tutorial
Cache-Control = max-age=2592000
Expires = Sat, 21 Aug 2021 18:08:17 GMT
Content-Length = 248
Keep-Alive =timeout=5, max=1500
Connection = Keep-Alive
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command line shows the user navigating to the Desktop directory and compiling a Java file named "HttpURLConnectionDemo.java". The output of the compilation includes the generated bytecode and various HTTP header information for the URL "https://www.javatpoint.com/java-tutorial". The headers listed include Date, Server, Location, Cache-Control, Expires, Content-Length, Keep-Alive, and Connection.

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac HttpURLConnectionDemo.java
C:\Users\WIN 8.1\Desktop>java HttpURLConnectionDemo
Date = Thu, 22 Jul 2021 18:08:17 GMT
Server = Apache
Location = https://www.javatpoint.com/java-tutorial
Cache-Control = max-age=2592000
Expires = Sat, 21 Aug 2021 18:08:17 GMT
Content-Length = 248
Keep-Alive = timeout=5, max=1500
Connection = Keep-Alive

C:\Users\WIN 8.1\Desktop>
```

[← Prev](#)[Next →](#)

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java InetAddress class

Java InetAddress class represents an IP address. The `java.net.InetAddress` class provides methods to get the IP of any host name *for example* `www.javatpoint.com`, `www.google.com`, `www.facebook.com`, etc.

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of `InetAddress` represents the IP address with its corresponding host name. There are two types of addresses: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

Moreover, `InetAddress` has a cache mechanism to store successful and unsuccessful host name resolutions.

IP Address

- An IP address helps to identify a specific resource on the network using a numerical representation.
- Most networks combine IP with TCP (Transmission Control Protocol). It builds a virtual bridge among the destination and the source.

There are two versions of IP address:

1. IPv4

IPv4 is the primary Internet protocol. It is the first version of IP deployed for production in the ARAPNET in 1983. It is a widely used IP version to differentiate devices on network using an addressing scheme. A 32-bit addressing scheme is used to store 2^{32} addresses that is more than 4 million addresses.

Features of IPv4:

- It is a connectionless protocol.
- It utilizes less memory and the addresses can be remembered easily with the class based addressing scheme.
- It also offers video conferencing and libraries.



2. IPv6

IPv6 is the latest version of Internet protocol. It aims at fulfilling the need of more internet addresses. It provides solutions for the problems present in IPv4. It provides 128-bit address space that can be used to form a network of 340 undecillion unique IP addresses. IPv6 is also identified with a name IPng (Internet Protocol next generation).

Features of IPv6:

- It has a stateful and stateless both configurations.
- It provides support for quality of service (QoS).
- It has a hierarchical addressing and routing infrastructure.

TCP/IP Protocol

- TCP/IP is a communication protocol model used connect devices over a network via internet.
- TCP/IP helps in the process of addressing, transmitting, routing and receiving the data packets over the internet.
- The two main protocols used in this communication model are:
 1. TCP i.e. Transmission Control Protocol. TCP provides the way to create a communication channel across the network. It also helps in transmission of packets at sender end as well as receiver end.
 2. IP i.e. Internet Protocol. IP provides the address to the nodes connected on the internet. It uses a gateway computer to check whether the IP address is correct and the message is forwarded correctly or not.

Java InetAddress Class Methods

Method	Description
public static InetAddress getByName(String host) throws UnknownHostException	It returns the instance of InetAddress containing LocalHost IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	It returns the instance of InetAddress containing local host name and address.
public String getHostName()	It returns the host name of the IP address.
public String getHostAddress()	It returns the IP address in string format.



Example of Java InetAddress Class

Let's see a simple example of InetAddress class to get ip address of www.javatpoint.com website.

InetDemo.java

```
import java.io.*;
import java.net.*;
public class InetDemo{
    public static void main(String[] args){
        try{
            InetAddress ip=InetAddress.getByName("www.javatpoint.com");

            System.out.println("Host Name: "+ip.getHostName());
            System.out.println("IP Address: "+ip.getHostAddress());
        }catch(Exception e){System.out.println(e);}
    }
}
```

Test it Now

Output:

```
Host Name: www.javatpoint.com
IP Address: 172.67.196.82
```

Program to demonstrate methods of InetAddress class

InetDemo2.java

```
import java.net.Inet4Address;
import java.util.Arrays;
import java.net.InetAddress;
public class InetDemo2
```

```
{  
public static void main(String[] args) throws Exception  
{  
InetAddress ip = InetAddress.getByName("www.javatpoint.com");  
InetAddress ip1[] = InetAddress.getAllByName("www.javatpoint.com");  
byte addr[]={72, 3, 2, 12};  
System.out.println("ip : "+ip);  
System.out.print("\nip1 : "+ip1);  
InetAddress ip2 = InetAddress.getByAddress(addr);  
System.out.print("\nip2 : "+ip2);  
System.out.print("\nAddress : " +Arrays.toString(ip.getAddress()));  
System.out.print("\nHost Address : " +ip.getHostAddress());  
System.out.print("\nisAnyLocalAddress : " +ip.isAnyLocalAddress());  
System.out.print("\nisLinkLocalAddress : " +ip.isLinkLocalAddress());  
System.out.print("\nisLoopbackAddress : " +ip.isLoopbackAddress());  
System.out.print("\nisMCGlobal : " +ip.isMCGlobal());  
System.out.print("\nisMCLinkLocal : " +ip.isMCLinkLocal());  
System.out.print("\nisMCNodeLocal : " +ip.isMCNodeLocal());  
System.out.print("\nisMCOrgLocal : " +ip.isMCOrgLocal());  
System.out.print("\nisMCSiteLocal : " +ip.isMCSiteLocal());  
System.out.print("\nisMulticastAddress : " +ip.isMulticastAddress());  
System.out.print("\nisSiteLocalAddress : " +ip.isSiteLocalAddress());  
System.out.print("\nhashCode : " +ip.hashCode());  
System.out.print("\n Is ip1 == ip2 : " +ip.equals(ip2));  
}  
}
```

Output:

```
C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac InetDemo2.java
C:\Users\WIN 8.1\Desktop>java InetDemo2
ip : www.javatpoint.com/172.67.196.82
ip1 : [Ljava.net.InetAddress;@7852e922
ip2 : /72.3.2.12
Address : [-84, 67, -60, 82]
Host Address : 172.67.196.82
isAnyLocalAddress : false
isLinkLocalAddress : false
isLoopbackAddress : false
isMCGlobal : false
isMClinkLocal : false
isMCNodeLocal : false
isMCOrgLocal : false
isMCsiteLocal : false
isMulticastAddress : false
isSiteLocalAddress : false
hashCode : -1404844974
Is ip1 == ip2 : false
C:\Users\WIN 8.1\Desktop>
```

In the above Java code, the various boolean methods of InetAddress class are demonstrated.

← Prev

Next →

AD

[For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

Java DatagramSocket and DatagramPacket

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming using the UDP instead of TCP.

Datagram

Datagrams are collection of information sent from one device to another device via the established network. When the datagram is sent to the targeted device, there is no assurance that it will reach to the target device safely and completely. It may get damaged or lost in between. Likewise, the receiving device also never know if the datagram received is damaged or not. The UDP protocol is used to implement the datagrams in Java.

Java DatagramSocket class

Java DatagramSocket class represents a connection-less socket for sending and receiving datagram packets. It is a mechanism used for transmitting datagram packets over network.

A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.

Commonly used Constructors of DatagramSocket class

- **DatagramSocket() throws SocketException:** it creates a datagram socket and binds it with the available Port Number on the localhost machine.
- **DatagramSocket(int port) throws SocketException:** it creates a datagram socket and binds it with the given Port Number.
- **DatagramSocket(int port, InetAddress address) throws SocketException:** it creates a datagram socket and binds it with the specified port number and host address.



Java DatagramSocket Class

Method	Description
void bind(SocketAddress addr)	It binds the DatagramSocket to a specific address and port.

void close()	It closes the datagram socket.
void connect(InetAddress address, int port)	It connects the socket to a remote address for the socket.
void disconnect()	It disconnects the socket.
boolean getBroadcast()	It tests if SO_BROADCAST is enabled.
DatagramChannel getChannel()	It returns the unique DatagramChannel object associated with the datagram socket.
InetAddress getInetAddress()	It returns the address to where the socket is connected.
InetAddress getLocalAddress()	It gets the local address to which the socket is connected.
int getLocalPort()	It returns the port number on the local host to which the socket is bound.
SocketAddress getLocalSocketAddress()	It returns the address of the endpoint the socket is bound to.
int getPort()	It returns the port number to which the socket is connected.
int getReceiverBufferSize()	It gets the value of the SO_RCVBUF option for this DatagramSocket that is the buffer size used by the platform for input on the DatagramSocket.
boolean isClosed()	It returns the status of socket i.e. closed or not.
boolean isConnected()	It returns the connection state of the socket.
void send(DatagramPacket p)	It sends the datagram packet from the socket.
void receive(DatagramPacket p)	It receives the datagram packet from the socket.

Java DatagramPacket Class

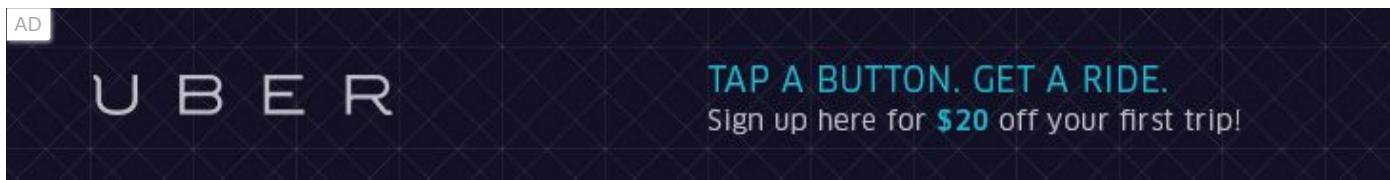
Java DatagramPacket is a message that can be sent or received. It is a data container. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

Commonly used Constructors of DatagramPacket class

- **DatagramPacket(byte[] barr, int length):** it creates a datagram packet. This constructor is

used to receive the packets.

- **DatagramPacket(byte[] barr, int length, InetAddress address, int port):** it creates a datagram packet. This constructor is used to send the packets.



Java DatagramPacket Class Methods

Method	Description
1) InetAddress getAddress()	It returns the IP address of the machine to which the datagram is being sent or from which the datagram was received.
2) byte[] getData()	It returns the data buffer.
3) int getLength()	It returns the length of the data to be sent or the length of the data received.
4) int getOffset()	It returns the offset of the data to be sent or the offset of the data received.
5) int getPort()	It returns the port number on the remote host to which the datagram is being sent or from which the datagram was received.
6) SocketAddress getSocketAddress()	It gets the SocketAddress (IP address + port number) of the remote host that the packet is being sent to or is coming from.
7) void setAddress(InetAddress iaddr)	It sets the IP address of the machine to which the datagram is being sent.
8) void setData(byte[] buff)	It sets the data buffer for the packet.
9) void setLength(int length)	It sets the length of the packet.
10) void setPort(int iport)	It sets the port number on the remote host to which the datagram is being sent.

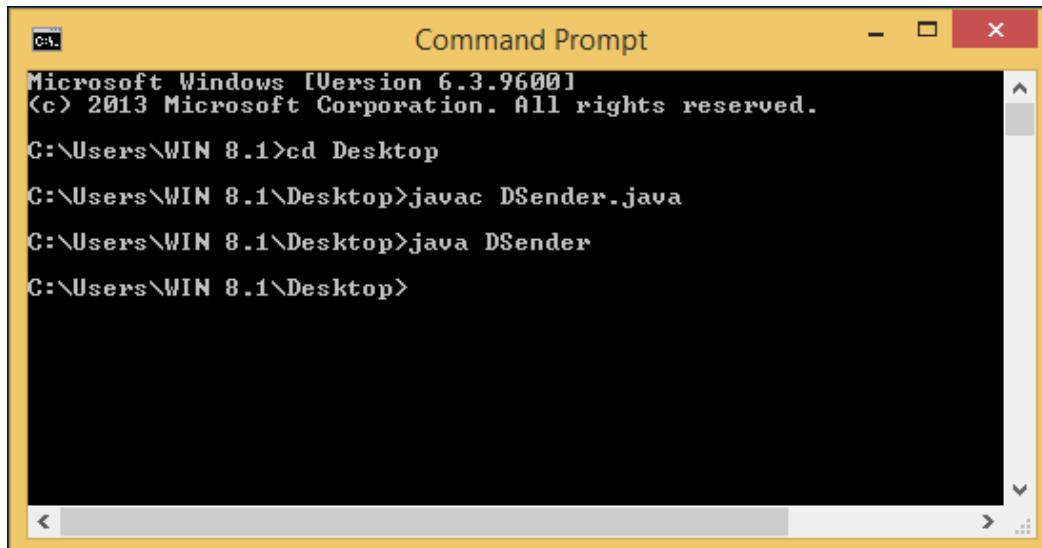
11) void setSocketAddress(SocketAddress addr)	It sets the SocketAddress (IP address + port number) of the remote host to which the datagram is being sent.
---	--

Example of Sending DatagramPacket by DatagramSocket

```
//DSender.java
import java.net.*;
public class DSender{
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket();
        String str = "Welcome java";
        InetAddress ip = InetAddress.getByName("127.0.0.1");

        DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);
        ds.send(dp);
        ds.close();
    }
}
```

Output:



The screenshot shows a Windows Command Prompt window titled 'Command Prompt'. The window displays the following command-line session:

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac DSender.java
C:\Users\WIN 8.1\Desktop>java DSender
C:\Users\WIN 8.1\Desktop>
```

Example of Receiving DatagramPacket by DatagramSocket

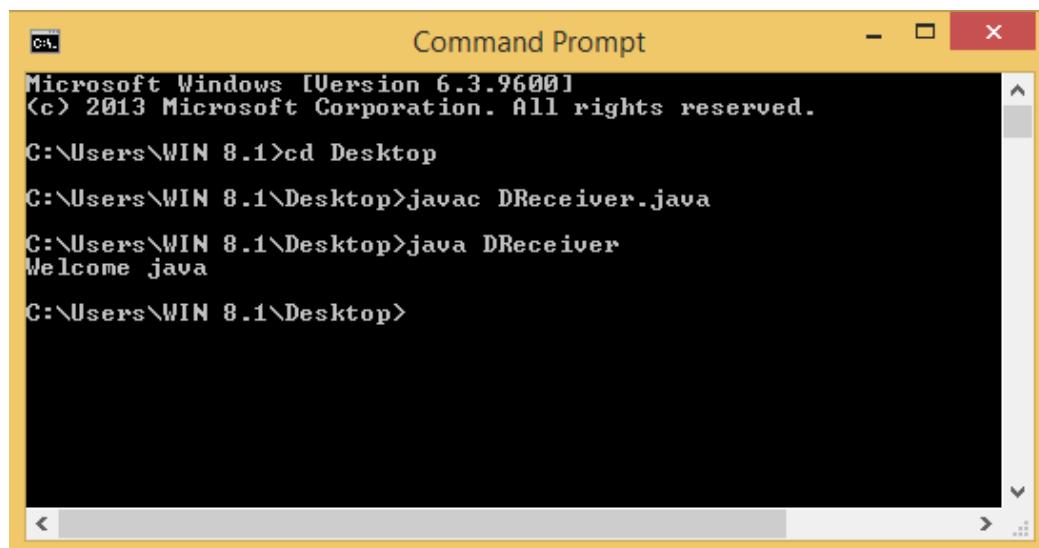
```
//DReceiver.java
```

```
import java.net.*;

public class DReceiver{

    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket(3000);
        byte[] buf = new byte[1024];
        DatagramPacket dp = new DatagramPacket(buf, 1024);
        ds.receive(dp);
        String str = new String(dp.getData(), 0, dp.getLength());
        System.out.println(str);
        ds.close();
    }
}
```

Output:



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a yellow border and a red close button. The text inside the window is as follows:

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\WIN 8.1>cd Desktop
C:\Users\WIN 8.1\Desktop>javac DReceiver.java
C:\Users\WIN 8.1\Desktop>java DReceiver
Welcome java

C:\Users\WIN 8.1\Desktop>
```

← Prev

Next →

Java Reflection API

Java Reflection is a process of examining or modifying the run time behavior of a class at run time.

The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The java.lang and java.lang.reflect packages provide classes for java reflection.

Where it is used

The Reflection API is mainly used in:

- IDE (Integrated Development Environment) e.g., Eclipse, MyEclipse, NetBeans etc.
- Debugger
- Test Tools etc.

Do You Know?

- How many ways can we get the instance of Class class?
- How to create the javap tool?
- How to create the appletviewer tool?
- How to access the private method from outside the class?

java.lang.Class class

The java.lang.Class class performs mainly two tasks:

- provides methods to get the metadata of a class at run time.
- provides methods to examine and change the run time behavior of a class.



Commonly used methods of Class class:

Method	Description
--------	-------------

1) public String getName()	returns the class name
2) public static Class forName(String className)throws ClassNotFoundException	loads the class and returns the reference of Class class.
3) public Object newInstance()throws InstantiationException,IllegalAccessException	creates new instance.
4) public boolean isInterface()	checks if it is interface.
5) public boolean isArray()	checks if it is array.
6) public boolean isPrimitive()	checks if it is primitive.
7) public Class getSuperclass()	returns the superclass class reference.
8) public Field[] getDeclaredFields()throws SecurityException	returns the total number of fields of this class.
9) public Method[] getDeclaredMethods()throws SecurityException	returns the total number of methods of this class.
10) public Constructor[] getDeclaredConstructors()throws SecurityException	returns the total number of constructors of this class.
11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException	returns the method class instance.

How to get the object of Class class?

There are 3 ways to get the instance of Class class. They are as follows:

- forName() method of Class class
- getClass() method of Object class
- the .class syntax

1) forName() method of Class class

- is used to load the class dynamically.
- returns the instance of Class class.

- o It should be used if you know the fully qualified name of class. This cannot be used for primitive types.

Let's see the simple example of `forName()` method.

FileName: Test.java

```
class Simple{}

public class Test{
    public static void main(String args[]) throws Exception {
        Class c=Class.forName("Simple");
        System.out.println(c.getName());
    }
}
```

Output:

```
Simple
```

2) `getClass()` method of Object class

It returns the instance of `Class` class. It should be used if you know the type. Moreover, it can be used with primitives.

FileName: Test.java

```
class Simple{}

class Test{
    void printName(Object obj){
        Class c=obj.getClass();
        System.out.println(c.getName());
    }
    public static void main(String args[]){
        Simple s=new Simple();

        Test t=new Test();
    }
}
```

```
t.printName(s);  
}  
}
```

Output:

```
Simple
```

3) The .class syntax

If a type is available, but there is no instance, then it is possible to obtain a Class by appending ".class" to the name of the type. It can be used for primitive data types also.

FileName: Test.java

```
class Test{  
    public static void main(String args[]){  
        Class c = boolean.class;  
        System.out.println(c.getName());  
  
        Class c2 = Test.class;  
        System.out.println(c2.getName());  
    }  
}
```

Output:

```
boolean  
Test
```

Determining the class object

The following methods of Class class are used to determine the class object:

- 1) **public boolean isInterface():** determines if the specified Class object represents an interface type.



2) public boolean isArray(): determines if this Class object represents an array class.

3) public boolean isPrimitive(): determines if the specified Class object represents a primitive type.

Let's see the simple example of reflection API to determine the object type.

FileName: Test.java

```

class Simple{}

interface My{



class Test{
    public static void main(String args[]){
        try{
            Class c=Class.forName("Simple");
            System.out.println(c.isInterface());

            Class c2=Class.forName("My");
            System.out.println(c2.isInterface());

        }catch(Exception e){System.out.println(e);}

    }
}
  
```

Output:



```

false
true
  
```

Pros and Cons of Reflection

Java reflection should always be used with caution. While the reflection provides a lot of advantages, it has some disadvantages too. Let's discuss the advantages first.

Pros: Inspection of interfaces, classes, methods, and fields during runtime is possible using reflection, even without using their names during the compile time. It is also possible to call methods, instantiate a class or to set the value of fields using reflection. It helps in the creation of Visual Development Environments and class browsers which provides aid to the developers to write the correct code.

Cons: Using reflection, one can break the principles of encapsulation. It is possible to access the private methods and fields of a class using reflection. Thus, reflection may leak important data to the outside world, which is dangerous. For example, if one access the private members of a class and sets null value to it, then the other user of the same class can get the NullReferenceException, and this behaviour is not expected.

Another demerit is the overhead in performance. Since the types in reflection are resolved dynamically, JVM (Java Virtual Machine) optimization cannot take place. Therefore, the operations performed by reflections are usually slow.

Conclusion

Because of the above-mentioned cons, it is generally advisable to avoid using reflection. It is an advanced feature that should only be used by programmers or developers who have a good knowledge of the basics of the language. Always remember! Whenever reflection is used, the security of the application is compromised.

Next Topics of Reflection API Tutorial

[newInstance\(\) method](#)

[Understanding javap tool](#)

[creating javap tool](#)

[creating appletviewer tool](#)

[Call private method from another class](#)

newInstance() method

The **newInstance()** method of **Class** class and **Constructor** class is used to create a new instance of the class.

The newInstance() method of Class class can invoke zero-argument constructor, whereas newInstance() method of Constructor class can invoke any number of arguments. So, Constructor class is preferred over Class class.

Syntax of newInstance() method of Class class

```
public T newInstance()throws InstantiationException,IllegalAccessException
```

Here T is the generic version. You can think of it as Object class. You will learn about generics later.

newInstance() Method Example-1

Let's see the simple example to use newInstance() method.

FileName: Test.java

```
class Simple{
    void message(){System.out.println("Hello Java");}
}

class Test{
    public static void main(String args[]){
        try{
            Class c=Class.forName("Simple");
            Simple s=(Simple)c.newInstance();
            s.message();

        }catch(Exception e){System.out.println(e);}

    }
}
```

Output:

```
Hello java
```

newInstance() method Example-2

We have learned in the introductory part of this topic that the newInstance() method of the Class class can only invoke the parameterless constructor. Let's understand the same with the help of an example.

FileName: ReflectionExample1.java

```
// important import statements
import static java.lang.System.out;
import java.lang.reflect.*;
import javax.swing.*;

public class ReflectionExample1
{
    // Allowing Java Virtual Machine to handle the ClassNotFoundException
    // main method
```

```

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

Object ob = null;
Class classDefinition = Class.forName("javax.swing.JLabel");
ob = classDefinition.newInstance();

// instance variable for holding the instance of the class
JLabel l1;

// checking whether the created object ob is
// the instance of JLabel or not.
// If yes, do the typecasting; otherwise, terminate the method
if(ob instanceof JLabel)
{
    l1 = (JLabel)ob;
}
else
{
    return;
}

// reaching here means the typecasting has been done
// now we can invoke the getText() method
out.println(l1.getText());

}
}

```

Output:

```

/ReflectionExample1.java:15: error: unreported exception InstantiationException; must be caught or declared to be thrown
    ob = classDefinition.newInstance();
               ^
Note: /ReflectionExample1.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
1 error

```

Explanation: The newInstance() method of the Class class only invokes the zero-argument constructor. However, we need to invoke the non-zero argument constructor for that, we need to use the newInstance() method of the class Constructor.

The following program shows how one can use the newInstance() method of the class Constructor to avoid the exception that has come in the above example.

FileName: ReflectionExample2.java

```

// important import statements
import static java.lang.System.out;
import java.lang.reflect.*;
import javax.swing.*;

public class ReflectionExample2
{

```

```
// main method
public static void main(String args[])
{
try
{
Class[] t = { String.class };
Class classDef = Class.forName("javax.swing.JLabel");

// getting the constructor
Constructor cons = classDef.getConstructor(t);

// setting the label
Object[] objct = { "My JLabel in Reflection."};

// getting the instance by invoking the correct constructor this time
Object ob = cons.newInstance(objct);

// instance variable for holding the instance of the class
JLabel l1;

// checking whether the created object ob is
// the instance of JLabel or not.
// If yes, do the typecasting; otherwise, exit from the method
if(ob instanceof JLabel)
{
l1 = (JLabel)ob;
}
else
{
// exiting from the method using the return statement
return;
}

// if the control reaches here, then it means the typecasting has been done
// now we can print the label of the JLabel instance
out.println(l1.getText());
}

// relevant catch block for handling the raised exceptions.
catch (InstantiationException ie)
{
out.println(ie);
}

catch (IllegalAccessException ie)
{
System.out.println(ie);
}

catch (InvocationTargetException ie)
{
out.println(ie);
}

catch (ClassNotFoundException e)
{
```

```

out.println(e);
}

catch (NoSuchMethodException e)
{
out.println(e);
}
}
}
}

```

Output:

My JLabel in Reflection.

<<Prev

Next>>

AD

[YouTube](#) For Videos Join Our YouTube Channel: [Join Now](#)

Feedback

- o Send your Feedback to feedback@javatpoint.com

Help Others, Please Share**Learn Latest Tutorials**

Splunk tutorial Splunk	SPSS tutorial SPSS	Swagger tutorial Swagger	T-SQL tutorial Transact-SQL	Tumblr tutorial Tumblr	React tutorial ReactJS
Regex tutorial Regex	Reinforcement learning tutorial Reinforcement Learning	R Programming tutorial R Programming	RxJS tutorial RxJS	React Native tutorial React Native	Python Design Patterns Python Design Patterns
Python Pillow tutorial Python Pillow	Python Turtle tutorial Python Turtle	Keras tutorial Keras			

Understanding javap tool

The **javap command** disassembles a class file. The javap command displays information about the fields, constructors and methods present in a class file.

Syntax to use javap tool

Let's see how to use javap tool or command.

```
javap fully_class_name
```

Example to use javap tool

```
javap java.lang.Object
```

Output:

```
Compiled from "Object.java"
public class java.lang.Object {
    public java.lang.Object();
    public final native java.lang.Class<?> getClass();
    public native int hashCode();
    public boolean equals(java.lang.Object);
    protected native java.lang.Object clone() throws java.lang.CloneNotSupportedException;
    public java.lang.String toString();
    public final native void notify();
    public final native void notifyAll();
    public final native void wait(long) throws java.lang.InterruptedException;
    public final void wait(long, int) throws java.lang.InterruptedException;
    public final void wait() throws java.lang.InterruptedException;
    protected void finalize() throws java.lang.Throwable;
    static {};
}
```

Another example to use javap tool for your class

Let's use the javap command for our java file.

FileName: Simple.java

```
class Simple{
    public static void main(String args[]){
        System.out.println("hello java");
    }
}
```

Now let's use the javap tool to disassemble the class file.

```
javap Simple
```

Output:

```
Compiled from "Simple.java"
class Simple {
    Simple();
    public static void main(java.lang.String[]);
}
```

javap -c command

You can use the `javap -c` command to see disassembled code. The code that reflects the java bytecode.

```
javap -c Simple
```

Output:

```
Compiled from "Simple.java"
class Simple {
    Simple();
    Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<":()V
        4: return

    public static void main(java.lang.String[]);
    Code:
        0: getstatic      #2                // Field java/lang/System.out:Ljava/io/PrintStream;
        3: ldc           #3                // String hello java
        5: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8: return
}
```

Options of javap tool

The important options of javap tool are as follows.

Option	Description
<code>-help</code>	prints the help message.
<code>-l</code>	prints line number and local variable
<code>-c</code>	disassembles the code
<code>-s</code>	prints internal type signature
<code>-sysinfo</code>	shows system info (path, size, date, MD5 hash)
<code>-constants</code>	shows static final constants

-version	shows version information
----------	---------------------------

Let's see how one can use these options with the help of an example. For the following file (ABC.java) we will use the above-mentioned options.

FileName: ABC.java

```
public class ABC
{
// main method
public static void main(String args[])
{
// declaring an integer array
int arr[] = {6, 7, 8, 6, 8, 0, 4};

// caculating size of the array
int size = arr.length;

// printing size of the array
System.out.println("The size of the array is " + size );

System.out.println("The 8th index of the array is " + arr[8] );

}
}
```

Command: javap -c ABC

Output:

```
Compiled from "ABC.java"
public class ABC {
    public ABC();
    Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<":()V
        4: return

    public static void main(java.lang.String[]);
    Code:
        0: bipush      7
        2: newarray     int
        4: dup
        5: iconst_0
        6: bipush      6
        8: iastore
        9: dup
       10: iconst_1
       11: bipush      7
       13: iastore
       14: dup
```

```

15:  iconst_2
16:  bipush      8
18:  iastore
19:  dup
20:  iconst_3
21:  bipush      6
23:  iastore
24:  dup
25:  iconst_4
26:  bipush      8
28:  iastore
29:  dup
30:  iconst_5
31:  iconst_0
32:  iastore
33:  dup
34:  bipush      6
36:  iconst_4
37:  iastore
38:  astore_1
39:  aload_1
40:  arraylength
41:  istore_2
42:  getstatic     #7           // Field java/lang/System.out:Ljava/io/PrintStream;
45:  iload_2
46:  invokedynamic #13,  0       // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
51:  invokevirtual #17          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
54:  getstatic     #7           // Field java/lang/System.out:Ljava/io/PrintStream;
57:  aload_1
58:  bipush      8
60:  iaload
61:  invokedynamic #23,  0       // InvokeDynamic #1:makeConcatWithConstants:(I)Ljava/lang/String;
66:  invokevirtual #17          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
69:  return
}

}

```

Command: javap -l ABC

Output:

```

Compiled from "ABC.java"
public class ABC {
    public ABC();
    LineNumberTable:
        line 1: 0

    public static void main(java.lang.String[]);
    LineNumberTable:
        line 6: 0
        line 9: 39
        line 12: 42
        line 14: 54
}

```

```
line 16: 69
}
```

Command: javap -s ABC



Output:

```
Compiled from "ABC.java"
public class ABC {
    public ABC();
    descriptor: ()V

    public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
}
```

Command: javap -sysinfo ABC

Output:

```
Classfile /C:/Users/Nikhil Kumar/Documents/ABC.class
Last modified Sep 11, 2021; size 970 bytes
SHA-256 checksum 576adf03386399a4691e0ce5b6c5aa5d964b082a1a61299bac5632942e413312
Compiled from "ABC.java"
public class ABC {
    public ABC();
    public static void main(java.lang.String[]);
}
```

Command: javap -constants ABC

Output:



```
Compiled from "ABC.java"
public class ABC {
    public ABC();
    public static void main(java.lang.String[]);
}
```

Command: javap -version ABC

Output:

```
14
Compiled from "ABC.java"
public class ABC {
```

```
public ABC();  
public static void main(java.lang.String[]);  
}
```

<<Prev

Next>>

AD



Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share



Learn Latest Tutorials

Splunk tutorial Splunk	SPSS tutorial SPSS	Swagger tutorial Swagger	T-SQL tutorial Transact-SQL	Tumblr tutorial Tumblr
React tutorial ReactJS	Regex tutorial Regex	Reinforcement learning tutorial Reinforcement Learning	R Programming tutorial R Programming	RxJS tutorial RxJS
React Native tutorial React Native	Python Design Patterns Python Design Patterns	Python Pillow tutorial Python Pillow	Python Turtle tutorial Python Turtle	Keras tutorial Keras

Preparation

Creating a program that works as javap tool

Following methods of **java.lang.Class** class can be used to display the metadata of a class.

Method	Description
public Field[] getDeclaredFields()throws SecurityException	returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
public Constructor[] getDeclaredConstructors()throws SecurityException	returns an array of Constructor objects reflecting all the constructors declared by the class represented by this Class object.
public Method[] getDeclaredMethods()throws SecurityException	returns an array of Method objects reflecting all the methods declared by the class or interface represented by this Class object.

Example of creating javap tool

Let's create a program that works like javap tool.

```
import java.lang.reflect.*;

public class MyJavap{
    public static void main(String[] args) throws Exception {
        Class c=Class.forName(args[0]);

        System.out.println("Fields.....");
        Field f[]=c.getDeclaredFields();
        for(int i=0;i<f.length;i++)
            System.out.println(f[i]);

        System.out.println("Constructors.....");
        Constructor con[]=c.getDeclaredConstructors();
        for(int i=0;i<con.length;i++)
            System.out.println(con[i]);

        System.out.println("Methods.....");
    }
}
```

```
Method m[]=c.getDeclaredMethods();
for(int i=0;i<m.length;i++)
    System.out.println(m[i]);
}
```

At runtime, you can get the details of any class, it may be user-defined or pre-defined class.

Output:

```
E:\>javac MyJavap.java
E:\>java MyJavap java.lang.Object
```

```
E:\>javac MyJavap.java
E:\>java MyJavap java.lang.Object
Fields.....
Constructors.....
public java.lang.Object()
Methods.....
protected void java.lang.Object.finalize() throws java.lang.Throwable
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public boolean java.lang.Object.equals(java.lang.Object)
public java.lang.String java.lang.Object.toString()
public native int java.lang.Object.hashCode()
public final native java.lang.Class java.lang.Object.getClass()
protected native java.lang.Object java.lang.Object.clone() throws java.lang.CloneNotSupportedException
private static native void java.lang.Object.registerNatives()
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()

E:\>
```

Creating your own appletviewer

As you know well that appletviewer tool creates a frame and displays the output of applet in the frame. You can also create your frame and display the applet output.

Example that works like appletviewer tool

Let's see the simple example that works like appletviewer tool. This example displays applet on the frame.

```
import java.applet.Applet;
import java.awt.Frame;
import java.awt.Graphics;

public class MyViewer extends Frame{
    public static void main(String[] args) throws Exception{
        Class c=Class.forName(args[0]);

        MyViewer v=new MyViewer();
        v.setSize(400,400);
        v.setLayout(null);
        v.setVisible(true);

        Applet a=(Applet)c.newInstance();
        a.start();
        Graphics g=v.getGraphics();
        a.paint(g);
        a.stop();

    }
}
```

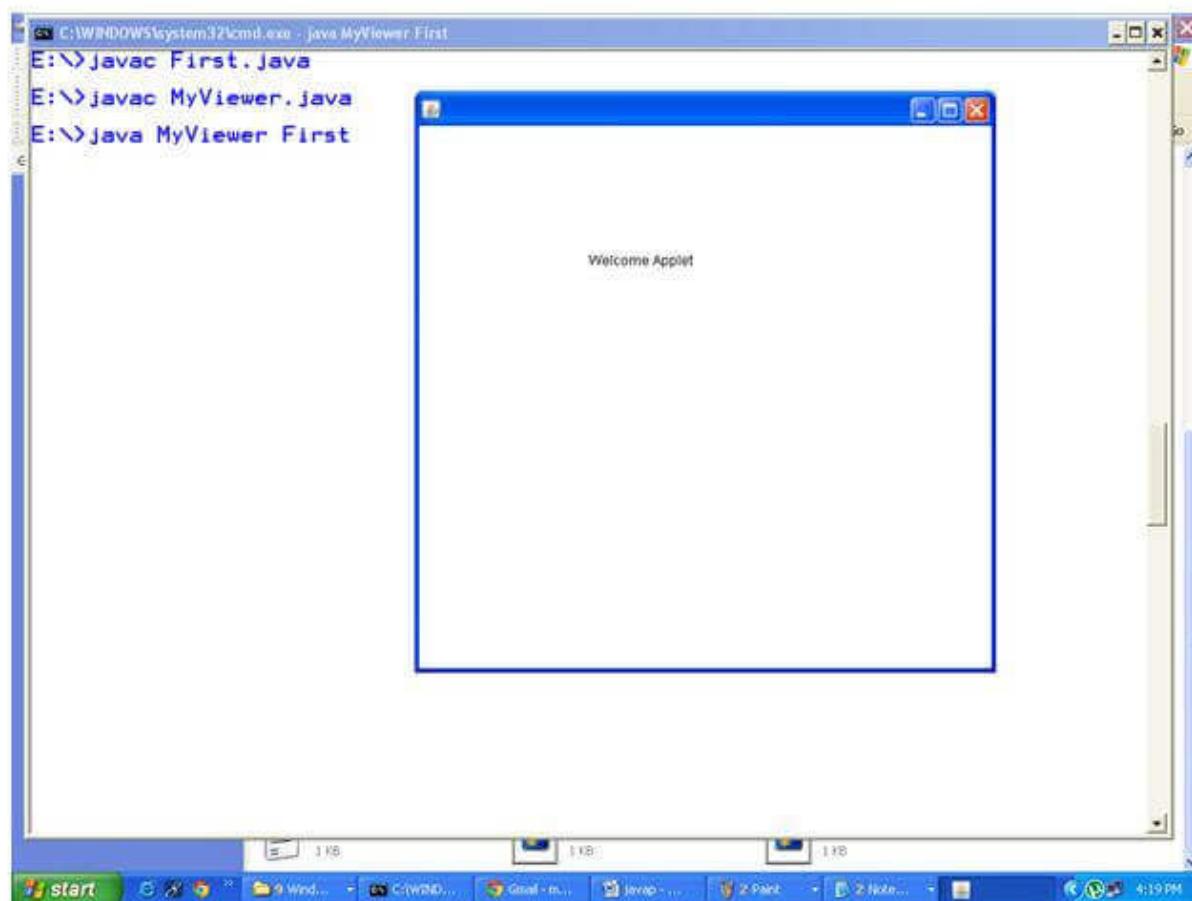
//simple program of applet

```
import java.applet.Applet;
```

```
import java.awt.Graphics;  
  
public class First extends Applet{  
  
    public void paint(Graphics g){g.drawString("Welcome",50, 50);}  
}
```

Output:

The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command line history shows the following steps:
E:\>javac First.java
E:\>javac MyViewer.java
E:\>java MyViewer First



How to call private method from another class in java

You can call the private method from outside the class by changing the runtime behaviour of the class.

With the help of **java.lang.Class** class and **java.lang.reflect.Method** class, we can call a private method from any other class.

Required methods of Method class

1) **public void setAccessible(boolean status) throws SecurityException** sets the accessibility of the method.

2) **public Object invoke(Object method, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException** is used to invoke the method.

Required method of Class class

1) **public Method getDeclaredMethod(String name, Class[] parameterTypes) throws NoSuchMethodException, SecurityException:** returns a Method object that reflects the specified declared method of the class or interface represented by this Class object.

Example of calling private method from another class

Let's see the simple example to call private method from another class.

File: A.java

```
public class A {  
    private void message(){System.out.println("hello java");}  
}
```

File: MethodCall.java

```
import java.lang.reflect.Method;  
public class MethodCall{  
    public static void main(String[] args) throws Exception{  
  
        Class c = Class.forName("A");  
        Object o= c.newInstance();  
    }  
}
```

```

Method m =c.getDeclaredMethod("message", null);
m.setAccessible(true);
m.invoke(o, null);
}
}

```

Output:

```
hello java
```

Another example to call parameterized private method from another class

Let's see the example to call parameterized private method from another class

File: A.java

```

class A{
private void cube(int n){System.out.println(n*n*n);}
}

```

File: M.java

```

import java.lang.reflect.*;
class M{
public static void main(String args[])throws Exception{
Class c=A.class;
Object obj=c.newInstance();

Method m=c.getDeclaredMethod("cube",new Class[]{int.class});
m.setAccessible(true);
m.invoke(obj,4);
}}

```

Output:

64

Accessing Private Constructors of a class

We know that constructors of a class are a special kind of method this is used to instantiate the class. To access the private constructor, we use the method `getDeclaredConstructor()`. The `getDeclaredConstructor()` is used to access a parameterless as well as a parametrized constructor of a class. The following example shows the same.

FileName: PvtConstructorDemo.java

```
// important import statements
import java.lang.reflect.Constructor;
import java.lang.reflect.Modifier;
import java.lang.reflect.InvocationTargetException;

class Vehicle
{
    // private fields of the class Vehicle
    private Integer vId;
    private String vName;

    // parameterless constructor
    private Vehicle()
    {
    }

    // parameterized constructor
    private Vehicle(Integer vId, String vName)
    {
        this.vId = vId;
        this.vName = vName;
    }

    // setter methods of the class Vehicle
```

```
public void setVehicleId(Integer vId)
{
    this.vId = vId;
}

public void setVehicleName(String vName)
{
    this.vName = vName;
}

// getter methods of the class Vehicle
public Integer getVehicleId()
{
    return vId;
}

public String getVehicleName()
{
    return vName;
}

public class PvtConstructorDemo
{
    // the createObj() method is used to create an object of
    // the Vehicle class using the parameterless constructor.

    public void craeteObj(int vId, String vName) throws InstantiationException, IllegalAccessException,
        IllegalArgumentException, InvocationTargetException, NoSuchMethodException
    {
        // using the parameterless contructor
        Constructor<Vehicle> constt = Vehicle.class.getDeclaredConstructor();

        constt.setAccessible(true);
        Object obj = constt.newInstance();
        if (obj instanceof Vehicle)
```

```
{  
    Vehicle v = (Vehicle)obj;  
    v.setVehicleId(vId);  
    v.setVehicleName(vName);  
    System.out.println("Vehicle Id: " + v.getVehicleId());  
    System.out.println("Vehicle Name: " + v.getVehicleName());  
}  
}  
  
// the craeteObjByConstructorName() method is used to create an object  
// of the Vehicle class using the parameterized constructor.  
public void craeteObjByConstructorName(int vId, String vName) throws NoSuchMethodException,  
InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException  
{  
  
    // using the parameterized contructor  
    Constructor<Vehicle> constt = Vehicle.class.getDeclaredConstructor(Integer.class, String.class);  
  
    if (Modifier.isPrivate(constt.getModifiers()))  
    {  
        constt.setAccessible(true);  
  
        Object obj = constt.newInstance(vId, vName);  
        if(obj instanceof Vehicle)  
        {  
            Vehicle v = (Vehicle)obj;  
            System.out.println("Vehicle Id: " + v.getVehicleId());  
            System.out.println("Vehicle Name: " + v.getVehicleName());  
        }  
    }  
}  
  
// delegating the responsibility to Java Virtual Machine (JVM) to handle the raised  
// exception  
// main method  
public static void main(String argvs[]) throws InstantiationException,
```

```
IllegalAccessException, IllegalArgumentException, InvocationTargetException,  
NoSuchMethodException, SecurityException  
{
```

```
// creating an object of the class PvtConstructorDemo  
PvtConstructorDemo ob = new PvtConstructorDemo();  
ob.craeteObj(20, "Indica");  
System.out.println(" ----- ");  
ob.craeteObjByConstructorName(30, "Alto");  
}  
}
```

Output:

```
Vehicle Id: 20  
Vehicle Name: Indica  
-----  
Vehicle Id: 30  
Vehicle Name: Alto
```

← Prev

Next →

AD

 YouTube For Videos Join Our YouTube Channel: Join Now

Java Regex

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

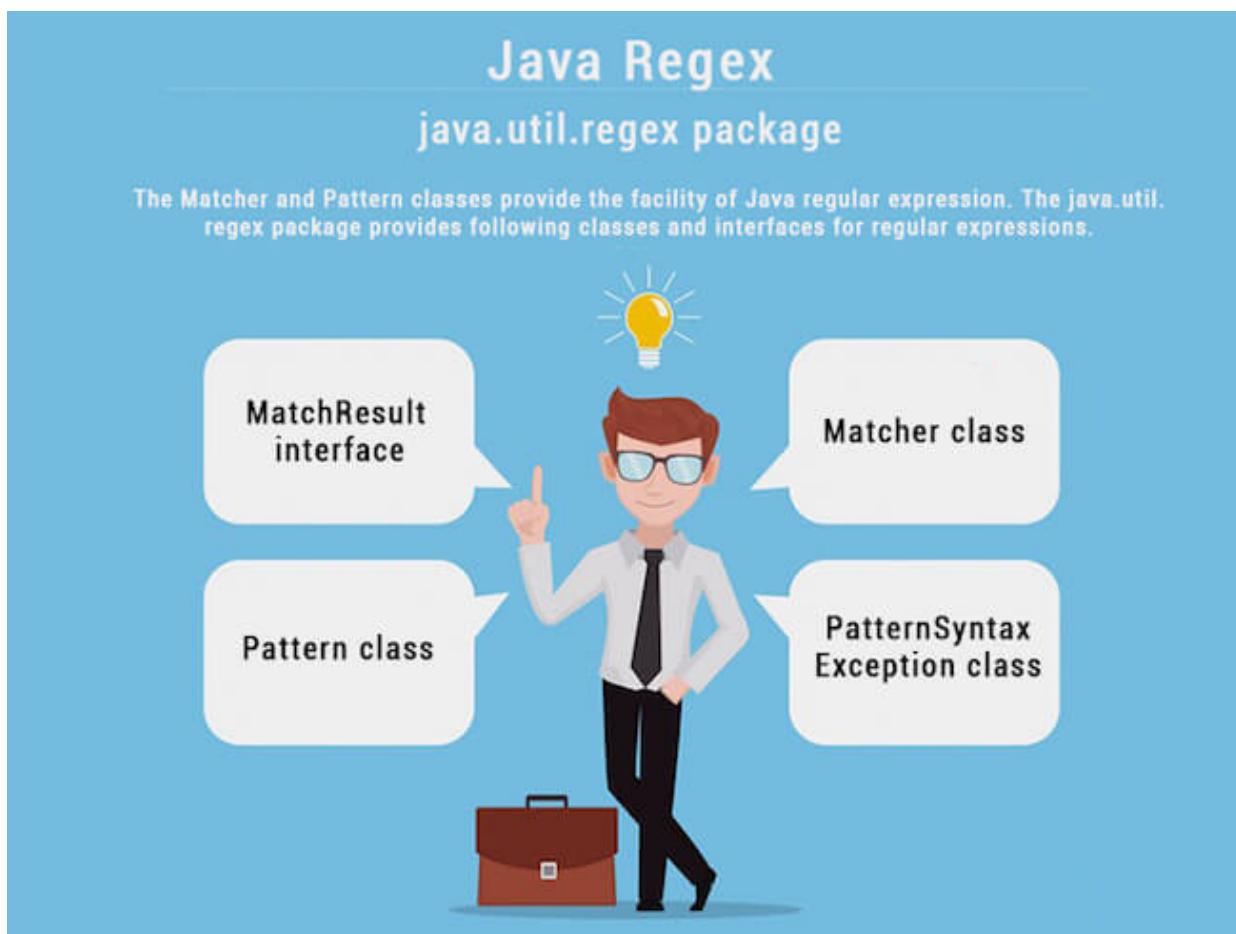
It is widely used to define the constraint on strings such as password and email validation. After learning Java regex tutorial, you will be able to test your regular expressions by the Java Regex Tester Tool.

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

java.util.regex package

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

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



Matcher class

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

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

Pattern class

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

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

Example of Java Regular Expressions

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

```
import java.util.regex.*;
public class RegexExample1{
public static void main(String args[]){
    //1st way
    Pattern p = Pattern.compile(".s");//. represents single character
    Matcher m = p.matcher("as");
    boolean b = m.matches();

    //2nd way
    boolean b2=Pattern.compile(".s").matcher("as").matches();

    //3rd way
    boolean b3 = Pattern.matches(".s", "as");

    System.out.println(b+ " "+b2+ " "+b3);
}}
```

Test it Now

Output

```
true true true
```

Regular Expression . Example

The . (dot) represents a single character.

```
import java.util.regex.*;
class RegexExample2{
public static void main(String args[]){
    System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)
    System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)
    System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)
}}
```

```
System.out.println(Pattern.matches(".s", "amms")); //false (has more than 2 char)
System.out.println(Pattern.matches(..s", "mas")); //true (3rd char is s)
{}
```

[Test it Now](#)



Regex Character classes

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

Regular Expression Character classes Example

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

[Test it Now](#)

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

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

Regular Expression Character classes and Quantifiers Example

```

import java.util.regex.*;
class RegexExample4{
public static void main(String args[]){
System.out.println("? quantifier ....");
System.out.println(Pattern.matches("[amn]?", "a")); //true (a or m or n comes one time)
System.out.println(Pattern.matches("[amn]?", "aaa")); //false (a comes more than one time)
System.out.println(Pattern.matches(
[amn]?", "aammmnn")); //false (a m and n comes more than one time)
System.out.println(Pattern.matches("[amn]?", "aazzta")); //false (a comes more than one time)
System.out.println(Pattern.matches("[amn]?", "am")); //false (a or m or n must come one time)

System.out.println("+ quantifier ....");
System.out.println(Pattern.matches("[amn]+", "a")); //true (a or m or n once or more times)
System.out.println(Pattern.matches("[amn]+", "aaa")); //true (a comes more than one time)
System.out.println(Pattern.matches(
[amn]+", "aammmnn")); //true (a or m or n comes more than once)
System.out.println(Pattern.matches("[amn]+", "aazzta")); //false (z and t are not matching pattern)

System.out.println("* quantifier ....");
}
}

```

```

System.out.println(Pattern.matches("
[amn]*", "ammmna")); //true (a or m or n may come zero or more times)

})

```

Test it Now

Regex Metacharacters

The regular expression metacharacters work as shortcodes.

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

Regular Expression Metacharacters Example

```

import java.util.regex.*;
class RegexExample5{
public static void main(String args[]){
System.out.println("metacharacters d...."); \\d means digit

System.out.println(Pattern.matches("\d", "abc")); //false (non-digit)
System.out.println(Pattern.matches("\d", "1")); //true (digit and comes once)
System.out.println(Pattern.matches("\d", "4443")); //false (digit but comes more than once)
}

```

```
System.out.println(Pattern.matches("\\d", "323abc")); //false (digit and char)
```

System.out.println("metacharacters D...."); \\D means non-digit

```
System.out.println(Pattern.matches("\\D", "abc")); //false (non-digit but comes more than once)
```

```
System.out.println(Pattern.matches("\\D", "1")); //false (digit)
```

```
System.out.println(Pattern.matches("\\D", "4443")); //false (digit)
```

```
System.out.println(Pattern.matches("\\D", "323abc")); //false (digit and char)
```

```
System.out.println(Pattern.matches("\\D", "m")); //true (non-digit and comes once)
```

System.out.println("metacharacters D with quantifier....");

```
System.out.println(Pattern.matches("\\D*", "mak")); //true (non-digit and may come 0 or more times)
```

```
}
```

[Test it Now](#)

Regular Expression Question 1

/*Create a regular expression that accepts alphanumeric characters only.

Its length must be six characters long only.*/

```
import java.util.regex.*;
class RegexExample6{
public static void main(String args[]){
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32")); //true
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32")); //false (more than 6 char)
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2")); //true
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2")); //false ($ is not matched)
}}
```

[Test it Now](#)

Regular Expression Question 2

/*Create a regular expression that accepts 10 digit numeric characters

starting with 7, 8 or 9 only.*/

```
import java.util.regex.*;
class RegexExample7{
public static void main(String args[]){
    System.out.println("by character classes and quantifiers ...");
    System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true
    System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true

    System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)
    System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)
    System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true

    System.out.println("by metacharacters ...");
    System.out.println(Pattern.matches("[789]{1}\\\d{9}", "8853038949"));//true
    System.out.println(Pattern.matches("[789]{1}\\\d{9}", "3853038949"));//false (starts from 3)

}}
```

Test it Now

Java Regex Finder Example

```
import java.util.regex.Pattern;
import java.util.Scanner;
import java.util.regex.Matcher;
public class RegexExample8{
    public static void main(String[] args){
        Scanner sc=new Scanner(System.in);
        while (true) {
            System.out.println("Enter regex pattern:");
            Pattern pattern = Pattern.compile(sc.nextLine());
            System.out.println("Enter text:");
            Matcher matcher = pattern.matcher(sc.nextLine());
            boolean found = false;
            while (matcher.find()) {
                System.out.println("I found the text "+matcher.group()+" starting at index "+
```

```
        matcher.start()+" and ending at index "+matcher.end());  
        found = true;  
    }  
    if(!found){  
        System.out.println("No match found.");  
    }  
}  
}  
}
```

Output:

```
Enter regex pattern: java  
Enter text: this is java, do you know java  
I found the text java starting at index 8 and ending at index 12  
I found the text java starting at index 26 and ending at index 30
```

← Prev

Next →

AD

 [For Videos Join Our Youtube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com

RMI (Remote Method Invocation)

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

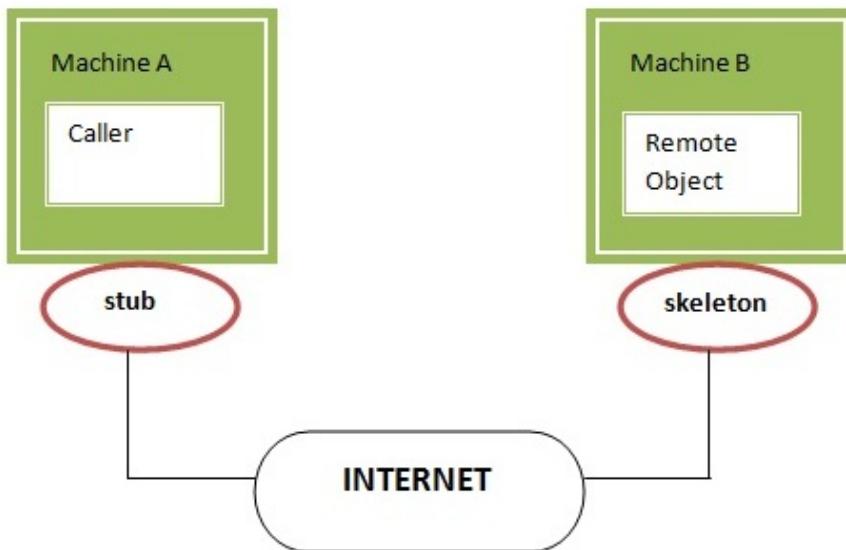
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.



Understanding requirements for the distributed applications

If any application performs these tasks, it can be distributed application.

1. The application need to locate the remote method
2. It need to provide the communication with the remote objects, and
3. The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

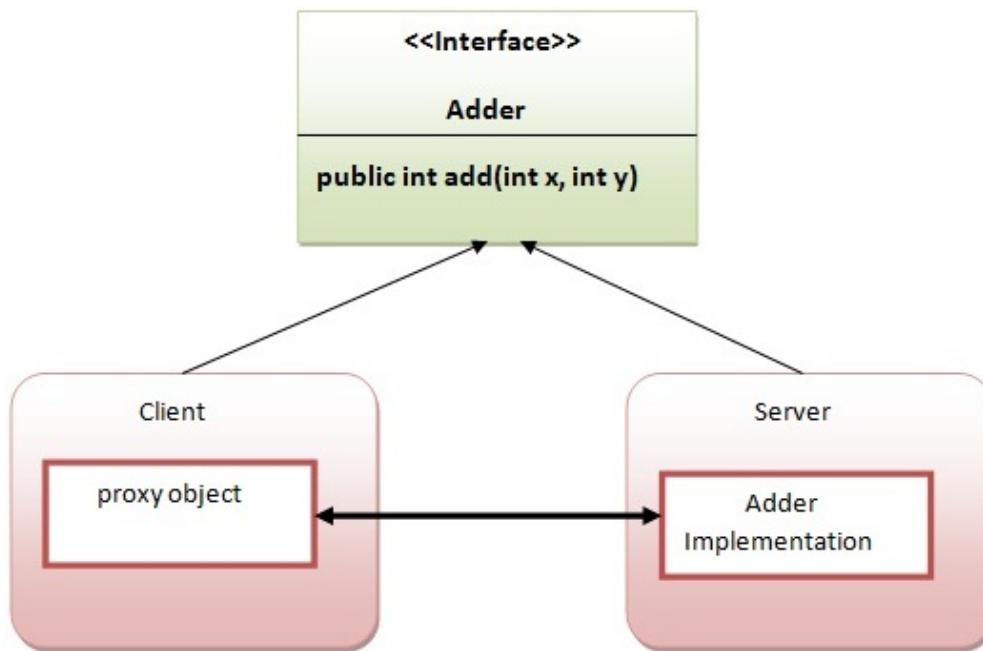
Java RMI Example

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

RMI Example

In this example, we have followed all the 6 steps to create and run the rmi application. The client application need only two files, remote interface and client application. In the rmi application, both client and server interacts with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.



1) create the remote interface

For creating the remote interface, extend the `Remote` interface and declare the `RemoteException` with all the methods of the remote interface. Here, we are creating a remote interface that extends the `Remote` interface. There is only one method named `add()` and it declares `RemoteException`.

```

import java.rmi.*;
public interface Adder extends Remote{
    public int add(int x,int y) throws RemoteException;
}
  
```

2) Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the `Remote` interface, we need to

- Either extend the `UnicastRemoteObject` class,

- o or use the `exportObject()` method of the `UnicastRemoteObject` class

In case, you extend the `UnicastRemoteObject` class, you must define a constructor that declares `RemoteException`.

```
import java.rmi.*;  

import java.rmi.server.*;  

public class AdderRemote extends UnicastRemoteObject implements Adder{  

    AdderRemote()throws RemoteException{  

        super();  

    }  

    public int add(int x,int y){return x+y;}  

}
```

3) create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The `rmic` tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```

4) Start the registry service by the rmiregistry tool

Now start the registry service by using the `rmiregistry` tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
rmiregistry 5000
```

5) Create and run the server application

Now rmi services need to be hosted in a server process. The `Naming` class provides methods to get and store the remote object. The `Naming` class provides 5 methods.

<code>public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;</code>	
--	--

It returns the reference of the remote object.

public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;	It binds the remote object with the given name.
public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;	It destroys the remote object which is bound with the given name.
public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;	It binds the remote object to the new name.
public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;	It returns an array of the names of the remote objects bound in the registry.

In this example, we are binding the remote object by the name sonoo.

```
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer{
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/sonoo",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.



```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
System.out.println(stub.add(34,4));
}catch(Exception e){}
}
}
```

download this example of rmi

For running **this** rmi example,

1) compile all the java files

```
javac *.java
```

2)create stub and skeleton object by rmic tool

```
rmic AdderRemote
```

3)start rmi registry in one command prompt

```
rmiregistry 5000
```

4)start the server in another command prompt

```
java MyServer
```

5)start the client application in another command prompt

```
java MyClient
```

Output of this RMI example

```
C:\Windows\system32\cmd.exe - rmiregistry 5000
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All Rights Reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>javac *.java
D:\Sonoo Programs\core\rmi\1>rmic AdderRemote
D:\Sonoo Programs\core\rmi\1>rmiregistry 5000
```

```
C:\Windows\system32\cmd.exe - java MyServer
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All Rights Reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyServer
```

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All Rights Reserved.

C:\Users\Sonoo>d:
D:\>cd "Sonoo Programs\core\rmi\1"
D:\Sonoo Programs\core\rmi\1>java MyClient
38
D:\Sonoo Programs\core\rmi\1>
```

Meaningful example of RMI application with database

Consider a scenario, there are two applications running in different machines. Let's say MachineA and MachineB, machineA is located in United States and MachineB in India. MachineB want to get list of all the customers of MachineA application.

Let's develop the RMI application by following the steps.

1) Create the table

First of all, we need to create the table in the database. Here, we are using Oracle10 database.

CUSTOMER400							
Table	Data	Indexes	Model	Constraints	Grants	Statistics	
UI Defaults	Triggers	Dependencies	SQL				
Query	Count Rows	Insert Row					
EDIT	ACC_NO	FIRSTNAME	LASTNAME	EMAIL	AMOUNT		
	67539876	James	Franklin	franklin1james@gmail.com	500000		
	67534876	Ravi	Kumar	ravimalik@gmail.com	98000		
	67579872	Vimal	Jaiswal	jaiswalvimal32@gmail.com	9380000		
row(s) 1 - 3 of 3							

[Download](#)

2) Create Customer class and Remote interface

File: *Customer.java*

```
package com.javatpoint;
public class Customer implements java.io.Serializable{
    private int acc_no;
    private String firstname, lastname, email;
    private float amount;
    //getters and setters
}
```

Note: Customer class must be Serializable.

File: *Bank.java*

```
package com.javatpoint;
import java.rmi.*;
import java.util.*;
interface Bank extends Remote{
public List<Customer> getCustomers()throws RemoteException;
}
```

3) Create the class that provides the implementation of Remote interface

File: BankImpl.java

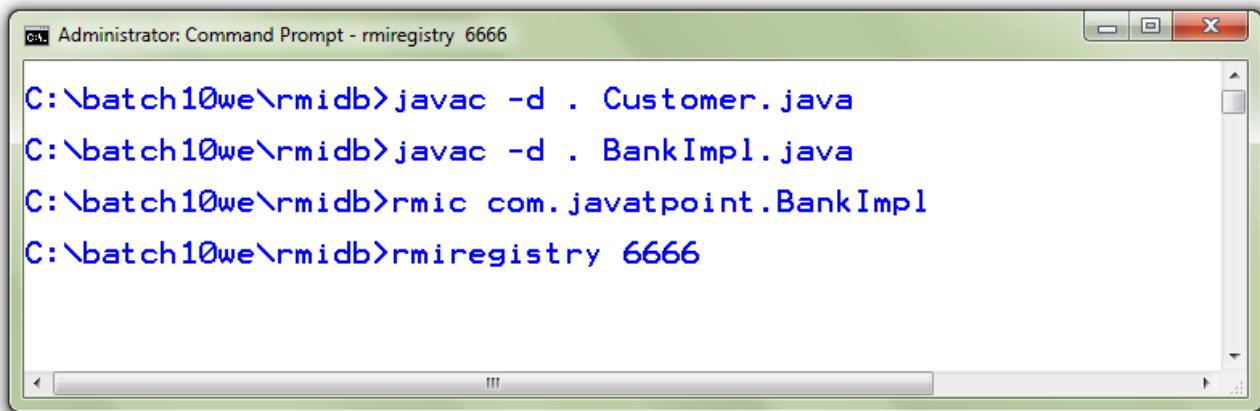
```
package com.javatpoint;
import java.rmi.*;
import java.rmi.server.*;
import java.sql.*;
import java.util.*;
class BankImpl extends UnicastRemoteObject implements Bank{
BankImpl()throws RemoteException{}

public List<Customer> getCustomers(){
List<Customer> list=new ArrayList<Customer>();
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","ora
PreparedStatement ps=con.prepareStatement("select * from customer400");
ResultSet rs=ps.executeQuery();

while(rs.next()){
Customer c=new Customer();
c.setAcc_no(rs.getInt(1));
c.setFirstname(rs.getString(2));
c.setLastname(rs.getString(3));
c.setEmail(rs.getString(4));
c.setAmount(rs.getFloat(5));
list.add(c);
}
}
```

```
con.close();
}catch(Exception e){System.out.println(e);}
return list;
}//end of getCustomers()
}
```

4) Compile the class rmic tool and start the registry service by rmiregistry tool



```
C:\batch10we\rmidb>javac -d . Customer.java
C:\batch10we\rmidb>javac -d . BankImpl.java
C:\batch10we\rmidb>rmic com.javatpoint.BankImpl
C:\batch10we\rmidb>rmiregistry 6666
```

5) Create and run the Server

File: MyServer.java

```
package com.javatpoint;
import java.rmi.*;
public class MyServer{
public static void main(String args[])throws Exception{
Remote r=new BankImpl();
Naming.rebind("rmi://localhost:6666/javatpoint",r);
}}
```

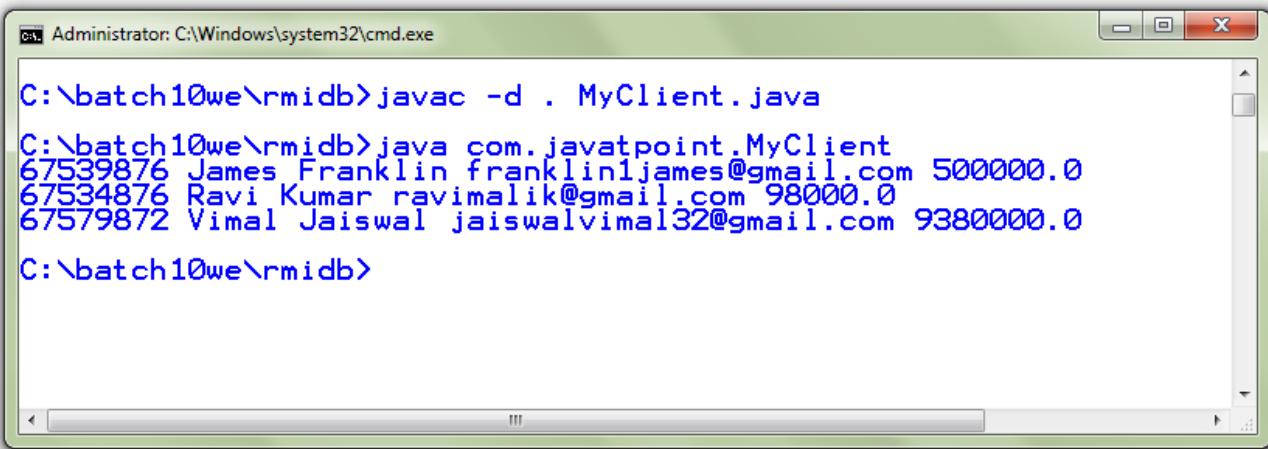
```
C:\batch10we\rmidb>javac -d . MyServer.java
C:\batch10we\rmidb>java com.javatpoint.MyServer
```

6) Create and run the Client

File: MyClient.java

```
package com.javatpoint;
import java.util.*;
import java.rmi.*;
public class MyClient{
    public static void main(String args[])throws Exception{
        Bank b=(Bank)Naming.lookup("rmi://localhost:6666/javatpoint");

        List<Customer> list=b.getCustomers();
        for(Customer c:list){
            System.out.println(c.getAcc_no()+" "+c.getFirstname()+" "+c.getLastname()
            +" "+c.getEmail()+" "+c.getAmount());
        }
    }
}
```



A screenshot of a Windows Command Prompt window titled "Administrator: C:\Windows\system32\cmd.exe". The window contains the following text:

```
C:\batch10we\rmidb>javac -d . MyClient.java
C:\batch10we\rmidb>java com.javatpoint.MyClient
67539876 James Franklin franklin1james@gmail.com 500000.0
67534876 Ravi Kumar ravimalik@gmail.com 98000.0
67579872 Vimal Jaiswal jaiswalvimal32@gmail.com 9380000.0
C:\batch10we\rmidb>
```

download this example of rmi with database

← Prev

Next →

AD

 [For Videos Join Our YouTube Channel: Join Now](#)

Feedback

- Send your Feedback to feedback@javatpoint.com