

9840195749

# JAVA Tutorial

HAARIS INFOTECH

SHOIAB MOHAMMAD

JAVA TUTORIAL

# Contents

---

## 1 INTRODUCTION TO JAVA 6

HISTORY OF JAVA	6
WHY TO LEARN JAVA?	7
JDK, JRE, JVM	11
FEATURES OF JAVA	14
INSTALLATION, HOW TO SETUP PATH?	19
APPLICATIONS OF JAVA	28
DATA TYPES AND TYPE CASTING	28
VARIABLES AND ARRAYS	31

## 2 OPERATORS AND STATEMENTS 0

OPERATORS	0
STATEMENTS	20
CONTROL STRUCTURES	22

## 3 OOPS 0

CLASSES, OBJECTS AND METHODS	7
MODIFIERS	13
INHERITANCE	18
CONSTRUCTORS	20
DESTRUCTORS	21

JAVA TUTORIAL

<b>ABSTRACTIONS</b>	<b>23</b>
<b>INNER CLASSES</b>	<b>25</b>
<b>PACKAGES</b>	<b>26</b>
<b>CLASSPATH</b>	<b>30</b>
<b>IMPORTING PACKAGES</b>	<b>31</b>
<b>INTERFACES</b>	<b>32</b>
<b>ENUMS</b>	<b>38</b>
<b>JAVA DATE AND TIME</b>	<b>42</b>
<b>METHODS AND METHOD OVERLOADING</b>	<b>44</b>
<b>SCOPE</b>	<b>51</b>

---

<b><i>4 FUNCTIONS</i></b>	<b><i>52</i></b>
---------------------------	------------------

<b>MATH CLASS</b>	<b>53</b>
<b>STRING</b>	<b>58</b>
<b>OBJECT CLONING</b>	<b>77</b>
<b>CALL BY VALUE</b>	<b>79</b>
<b>COMMAND LINE ARGUMENTS</b>	<b>81</b>
<b>COLLECTIONS</b>	<b>83</b>
<b>DATA STRUCTURES</b>	<b>89</b>
<b>JAVA GENERICS</b>	<b>134</b>

---

<b><i>5 EXCEPTION HANDLING</i></b>	<b><i>137</i></b>
------------------------------------	-------------------

<b>JAVA EXCEPTIONS</b>	<b>137</b>
<b>TYPES OF EXCEPTION</b>	<b>139</b>
<b>REGULAR EXCEPTIONS</b>	<b>151</b>

**JAVA TUTORIAL**

<b>FLAGS</b>	<b>152</b>
<b>EXPRESSION PATTERNS</b>	<b>155</b>
<b>METACHARACTERS</b>	<b>161</b>
 <b><u>6 THREADS</u></b>	 <b><u>163</u></b>
 SYNCHRONIZATION	 176
MULTITHREADING	181
NEED FOR MULTITHREADING	181
WHAT IS MULTITASKING	182
CONCURRENCY	184
 <b><u>7 FILE INPUT &amp; OUTPUT</u></b>	 <b><u>188</u></b>
 JAVA FILES INPUT AND OUTPUT	 188
STREAM	188
FILE STREAMS	189
BUFFERED STREAM	202
JAVA NETWORKING	221
	245
 <b><u>8 JAVA AWT</u></b>	 <b><u>249</u></b>
 BASICS OF AWT	 249
EVENT HANDLING	252
AWT CONTROLS	257

**JAVA TUTORIAL**

DESIGN PROGRAMMING CONCEPT	260
LISTENERS	265
ADAPTER CLASSES	271
CLOSING AWT WINDOW	272
BUTTON, LABEL, TEXTFIELD, TEXTAREA, CHECKBOX, CHECKBOXGROUP	272
CHOICE, LIST, CANVAS, SCROLLBAR, MENU, MENUITEM POPUP MENU, PANEL, DIALOG, TOOLKIT, DEMO ON EVENT HANDLING WITH AWT CONTROLS	285
EVENT HANDLING WITH AWT	305
<b><u>9 SWING</u></b>	<b><u>315</u></b>
JAVA SWING	315
DIFFERENCE BETWEEN AWT & SWING	317
SWING CONTROLS	320
JAVA SWING APPS	324
LAYOUT MANAGER	326
GRAPHICS IN SWING	331
DISPLAYING IMAGES	334
<b><u>10 JDBC</u></b>	<b><u>337</u></b>
INTRODUCTION	337
ARCHITECTURE & QUERYING WITH JDBC	339
QUERYING WITH JDBC	342
THE <i>DATABASEMETADATA</i> OBJECT	363

SHOIB MOHAMMAD

9840195749

## JAVA TUTORIAL

THE <i>RESULTSETMETADATA</i> OBJECT	365
MAPPING DATABASE TYPES TO JAVA TYPES	367
THE <i>PREPAREDSTATEMENT</i> OBJECT	369
THE <i>CALLABLESTATEMENT</i> OBJECT	372
USING <i>TRANSACTIONS</i>	375
SUMMARY OF JDBC CLASSES	379

## JAVA TUTORIAL

# 1 *Introduction to JAVA*

---

This section introduces you with **JAVA**. Java is an Object-Oriented Programming similar to C++ and Smalltalk. But JAVA is platform independent and follows the principle of WORA (Write Once and Run Anywhere). It is unlike C, C++ which means that you write a JAVA program and compile it only once and run it on any operation system be it Windows, Linux, Solaris etc.

If we talk about C++ then it is not all platform independent because needs to be compiled on each operation system in order to execute the program on different operation system. We will discuss this in detail in section **Java Virtual Machine** that how it achieves such a platform independence whereas C, C++, FORTRAN, VB, VC++ and many others.

## History of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak” but was renamed to “Java” in 1995 due to a patent search which determined that the name was copyrighted and used for another programming language. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of

## JAVA TUTORIAL

1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Java has travelled a long way and it has made a mark of presence in 21st Century. Today, Java is used for many types of applications like embedded applications, financial applications, desktop applications, flight simulators, Image Processors, Games, distribute enterprise applications called J2EE and many more.

## Why to learn JAVA?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming languages in the world
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has a huge community support (tens of millions of developers)
- Java is an object-oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs

## JAVA TUTORIAL

- As Java is close to C++ and C#, it makes it easy for programmers to switch to Java or vice versa

### **Java is easy to learn:**

Java is quite easy to learn and can be understood in a short span of time as it has a syntax similar to English.

### **Java has a large community:**

There is a large online community of Java users ranging from beginner, advanced and even expert levels that are particularly helpful in case any support is required.

### **Java has an abundant API:**

Java has an abundant Application Programming Interface (API) that includes many Java classes, packages, interfaces, etc. This is useful for constructing applications without necessarily knowing their inside implementations.

Java has mainly three types of API i.e., **Official Java core API's**, **Optional official Java API's** and **Unofficial API's**. These APIs overall are used for almost everything including networking, I/O, databases, media, XML parsing, speech synthesis, etc.

### **Java has multiple Open-Source Libraries:**

## JAVA TUTORIAL

Open-source libraries have resources that can be copied, studied, changed, shared, etc. There are multiple open-source libraries in Java such as **JHipster**, **Maven**, **Google Guava**, **Apache Commons**, etc. that can be used to make Java development easier, cheaper and faster.

### **Java has Powerful Development Tools:**

There are many Integrated development environments (IDE's) in Java that provides various facilities for software development to programmers. Powerful Java IDE's such as **Eclipse**, **NetBeans**, **IntelliJ IDEA**, etc. play a big role in the success of Java.

These IDEs provide many facilities such as debugging, syntax highlighting, code completion, language support, automated refactoring, etc. that make coding in Java easier and faster. Java has created a base for the Android operating system and opted around 90% fortune 500 companies for develop a lot of back-end applications. Also, it plays a great role in Apache Hadoop data processing, Amazon Web Services, and Windows Azure, etc.

### **Java is Free of Cost:**

One of the reasons Java is very popular among individual programmers is that it is available under the Oracle Binary Code License (BCL) free of charge. This means that Java is free for development and test environments, but for commercial purposes, a small fee is required.

## JAVA TUTORIAL

### **Java is Platform Independent:**

Java is platform-independent as the Java source code is converted to byte code by the compiler which can then be executed on any platform using the Java Virtual Machine. Java is also known as a WORA (write once, run anywhere) language because it is platform-independent.

Also, the development of most Java applications occurs in a Windows environment while they are run on a UNIX platform because of the platform-independent nature of Java.

### **Java has great Documentation Support:**

The documentation support for Java is excellent using Javadoc which is the documentation generator for Java. It uses the Java source code to generate the API documentation in HTML format. So, Javadoc provides a great reference while coding in Java so that understanding the code is quite simple.

### **Java is Versatile:**

Java is very versatile as it is used for programming applications on the web, mobile, desktop, etc. using different platforms. Also, Java has many features such as dynamic coding, multiple security features, platform-independent characteristics, network-centric designing, etc. that make it quite versatile.

## JAVA TUTORIAL

# JDK, JRE, JVM

### *Java Virtual Machine (JVM)*

Java Virtual Machine is a small application mainly written in C language that needs to be installed in order to execute JAVA programs. Whenever a Java program is compiled it is converted into **BYTE CODE**. Further when java program is executed, JVM executes those byte codes and converts them into machine language understood by the underlying operating system. Since JVM is platform dependent it makes Java platform independent. Byte codes produced during compilation are always same for all operating systems but since JVM is platform dependent it reads those byte codes and convert them into the machine language for which the JVM has been designed. e.g. JVM built for Linux operating system will convert the byte codes into machine language understood by Linux and the underlying machine hardware.

Most modern languages are designed to be compiled, not interpreted, mostly out of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet.

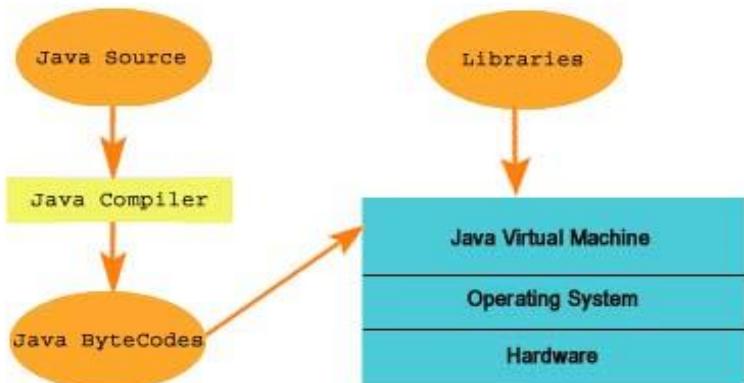
If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU and operating system connected to the Internet, which

## JAVA TUTORIAL

is not a feasible solution. Thus, the implementation of bytecode is the easiest way to create truly portable programs. Interpreted Java code also helps to make it secure. Since the execution of a Java program is under the control of the JVM, the JVM can control the program, and prevent it from generating side effects outside of it. This makes it ideal for applets to be downloaded from across the Internet, and executed on any CPU and Operating System that has the corresponding JVM implementation.

**Java Development Kit (JDK)** is a software development tool used for developing Java programs. It works as a compiler and a debugging tool to develop applets and applications. JDK is a set of development tools such as JavaDoc, Java Debugger, etc., and JRE to execute the program. It is platform dependent and used by Java developers. JDK can be used in Windows, MacOS systems.

**Java Runtime Environment (JRE)** is a software tool that provides tools which are necessary for the software to execute Java applications. JRE consists of class libraries, JVM, loader class. JRE comes along with JDK tools and need not to install JRE separately. JRE is used by those who want to run the Java Programs i.e., end users of your system. It is important to use JRE if we have to run Java applets.

**JAVA TUTORIAL*****Java in Time Compiler (JIT)***

Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code. Sun provides its Just In Time (JIT) compiler for bytecode which when included in the JVM, it compiles bytecode into executable code in real time one by one as the program demands. The JIT compiles code as it is needed during execution. However, the just-in-time approach still yields a significant performance boost. Whether your Java program is actually taken in the traditional way or compiled on the fly its functionality is the same.

Suppose there are two files (Java class) which are used by a Java program will be loaded into the memory one by one on a

SHOIB MOHAMMAD

9840195749

## JAVA TUTORIAL

demand basis. JVM will not check for the accessibility of the files before executing the code. In case if it is not able to find the required file in between the execution of the program, JVM will throw an exception saying that it has not found the file and exit the program from that point. The program will not complete successfully.

Although the other programming languages which existed before the origin of Java were as good and user friendly to the professional programmers, they still expected something advance to come up with all those features which were definitely the cause of worry to them with respect to the security of their code and this thought gave birth to a revolution which discovered another Programming Language-JAVA with the features to ensure the security and the portability of their programs.

## Features of JAVA

Developing your applications using the Java programming language results in software that is portable across multiple machine architectures, operating systems, and graphical user interfaces, secure, and high performance.

Not only the security, but there were few more areas which were taken care by this Programming Language normally identified as the **JAVA-BUZZWORDS**. These words define the additional features and considerations which gave JAVA-

## JAVA TUTORIAL

complete success and acceptance from the programmers from the software world.

Together, the above requirements comprise quite a collection of buzzwords. So, let's examine some of them and their benefits before going on.

- Simple and Object oriented
- Secure and Portable
- Robust
- Multithreaded and Architecture-neutral
- Interpreted and High performance
- Distributed and Dynamic

Although these words are self-explanatory in them. We will take a look on each one of them in our tutorial below.

### *Simple and Object-Oriented*

As mentioned above while discussing the overview of Java we understood that Java got its origin as a result of the deep study of the pre-existing Programming languages like C and C++. It makes very easy for any professional programmer to have a clear understanding of Java and its functionality if he has basic knowledge of C++ and the OOPS (Object Oriented programming) concepts. And hence the fundamental concepts of Java technology can be grasped quickly and the programmers can be productive from the very beginning as its look and feel makes it comfortable to the programmers as the

## JAVA TUTORIAL

beginners of Java Too. Not only this, the Founders of JAVA made sure that although Java is originated from the basics of the pre-existing Programming Languages it still avoids the features of those languages which were confusing and were not accepted happily by the Professionals. The Java programming language is designed to be object oriented from the ground up. After thirty years of regular exercise finally the Object technology got the acceptance in the programming mainstream. The Object-Oriented Concepts made it possible to function within increasingly complex, network-based environments, and so it can be concluded that Java technology provides a clean and efficient object-based development platform to the programmers.

### *Secure & Portable*

As discussed previously, under the head JVM (Java Virtual machine), the output which we get from the Java Complier is not in the form of directly ‘Executable Code’ but it is in the form of BYTE CODES., Byte code is nothing but the set of instructions which are executed by the Java Run time Environment, the JVM. Irrespective of all the other programming languages which are complied, java is actually interpreted by the JVM and hence makes it very much safe and secure to be downloaded through the internet as JVM makes sure that the Java Program is in its complete control and do not affect anything outside its environment. Since the code gets converted into the Byte Code by the JVM, it gives huge amount

## JAVA TUTORIAL

of portability to Java as it can run on any platform in any environment provided, they have the JVM available on it. The Byte codes will always be the same in spite of having variations in the JVM with respect to the environment and this is one of the other very strong supportive features of JAVA.

### ***Robust***

Java is considered to be a very robust Programming language as it ensures that the Java Programs run easily and successfully on the variety of environments and platforms. Since Java is a strictly typed language it helps the programmer to reduce the chances of making the run time errors and also handling them within the programs itself which normally cannot happen in other programming languages.

### ***Multi-threaded and Architecture Neutral***

Multithreading is another feature of Java which lets programmer opt for Java as the programming language as it facilitates the programmer to do various tasks within one program itself without having issues of conflicts between the tasks. As it has been discussed earlier in the sections above, that Java Program gets converted into Byte Code which is very much platform independent it makes Java as highly Architecture-neutral. And hence it can be stated that Java has successfully achieved “write once; run anywhere, anytime, forever.” which was the basic goal of the inventors of Java Programming Language.

## JAVA TUTORIAL

### ***Interpreted and High Performance***

Java programs get converted into Byte code which is easily interpreted by the JVM on any machine and in any environment and although it gets interpreted by the JVM it is not the case that the Java Code can't be complied into the native machine code and it is taken care by the JIT (Just –In-Time) Complier which is available with the JVM and thus ensures very high Performance of the language.

### ***Distributed and Dynamic***

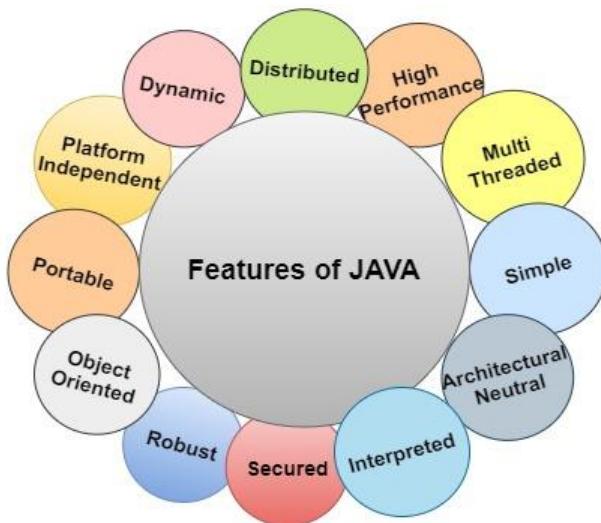
Since, Java was designed with the aim of making it accessible over the internet, providing it the feature of being called as Distributed programming language was very important as it had to handle the TCP/IP protocols of the internet. Java technology and its runtime environment have combined them to produce a flexible and powerful programming system.

Since the byte codes are loaded in small application forms as applets it becomes mandatory for Java programs to carry the sufficient amount of information related to the objects which are executed at the run time so as to provide verification with respect to that particular object hence giving it the feature of being called as Dynamic Language too.

These above-mentioned features can be easily termed as the major wings of the Java Programming language which has

## JAVA TUTORIAL

given it the open acceptance of the professions of the programming World.



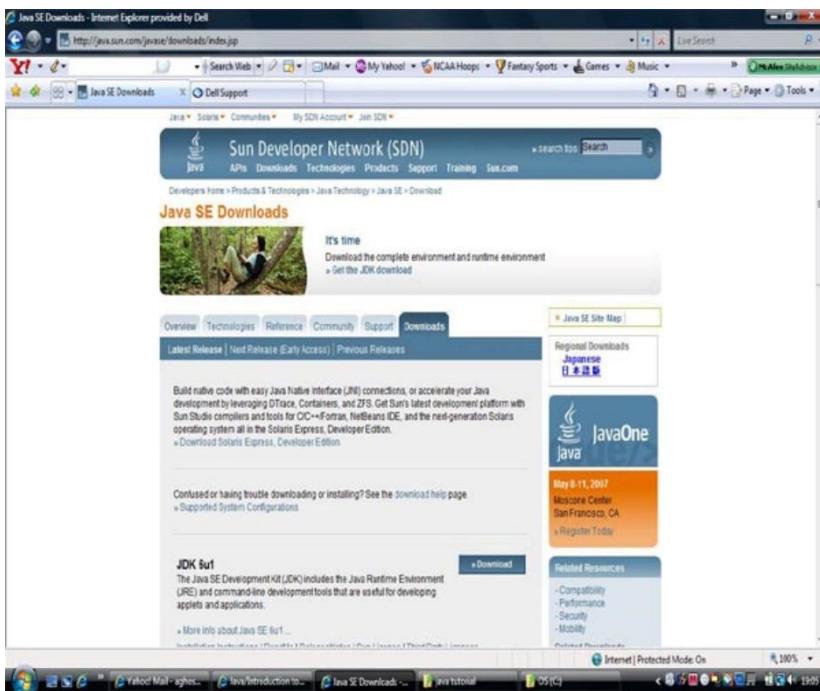
## Installation, How to setup path?

In order to execute java programs and application one needs to install Java Runtime Environment that has all supporting tools like java compiler, java debugger, tools to execute java programs. In order to install JDK (Java Development Toolkit) on the computer you need to download the JDK from Java Sun Microsystems.

Please follow the step to download JDK 1.5 from Sun Microsystems for Windows platform.

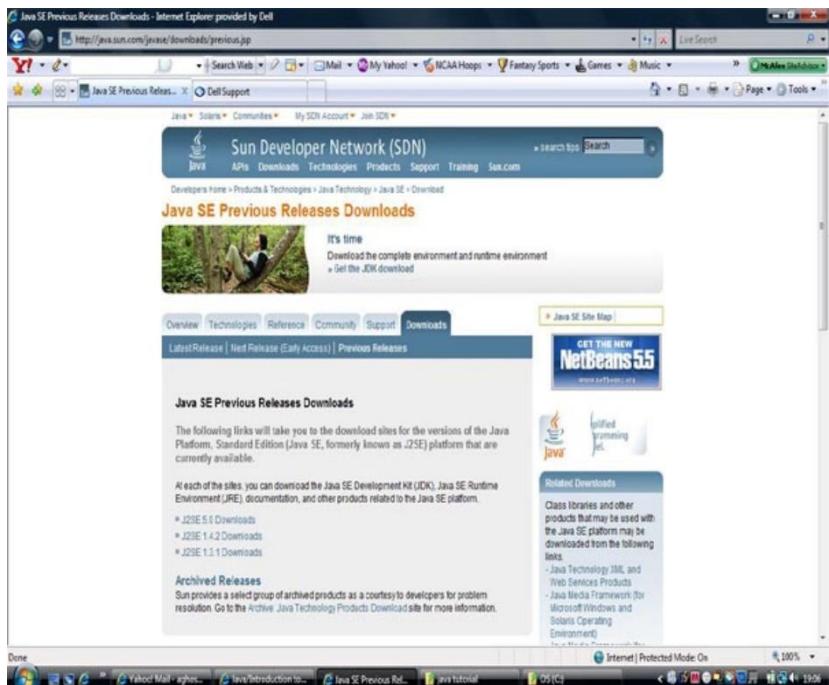
## JAVA TUTORIAL

1. You will find a screen like below.



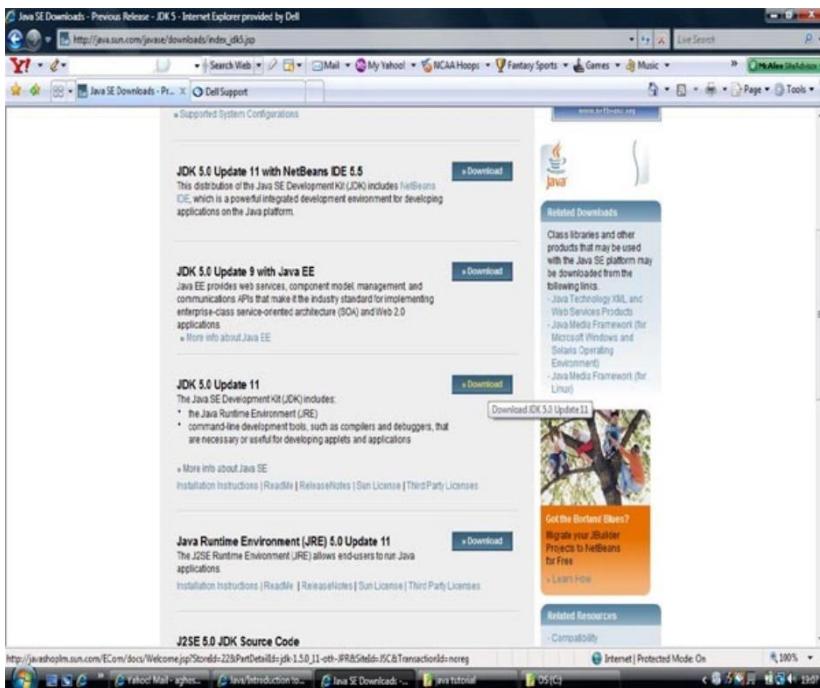
2. Click on the link called "Previous Releases". Then click on "J2SE 5.0 Downloads"

## JAVA TUTORIAL



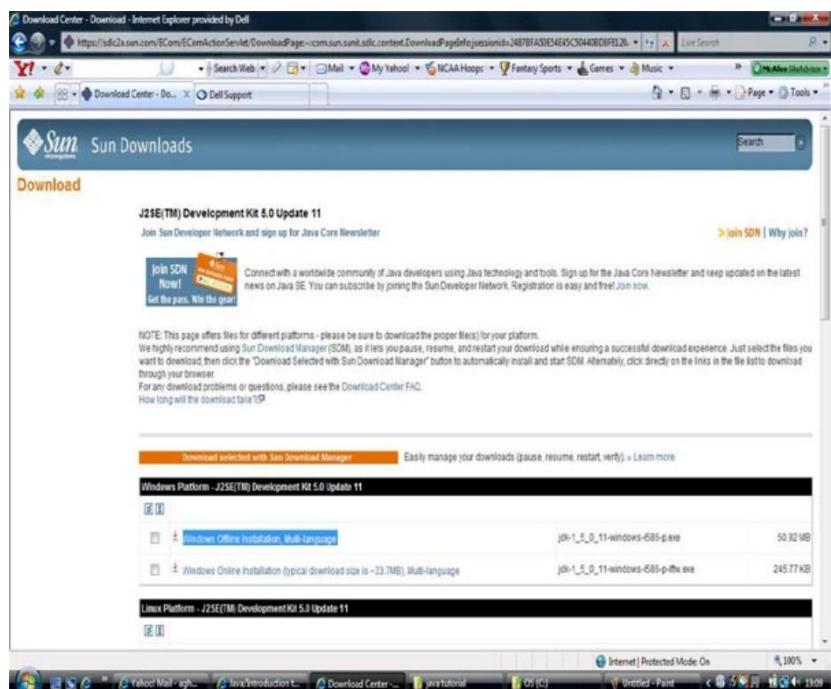
3. Click on the link "JDK 5.0 Update 11".

## JAVA TUTORIAL



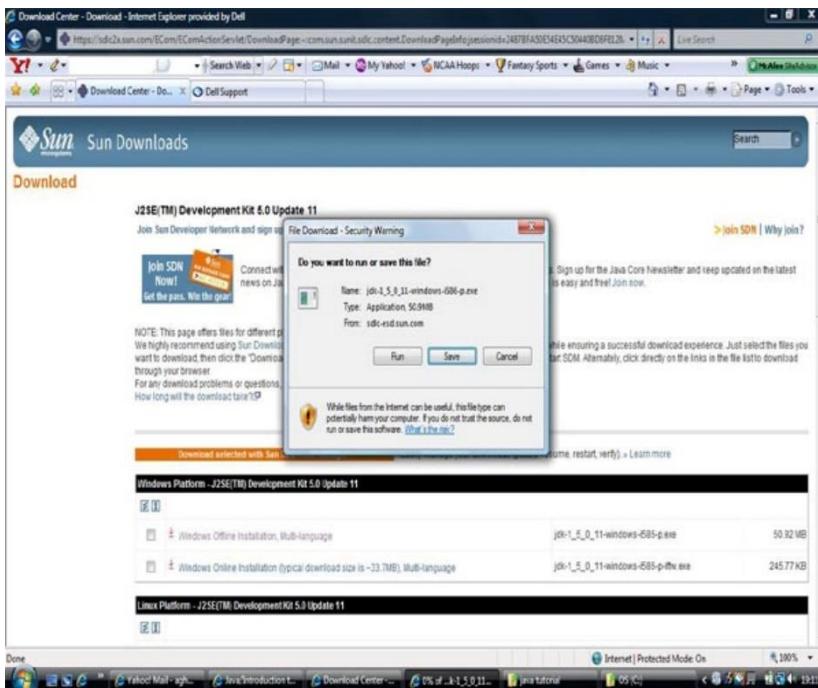
4. Click on the radio button with label "Accept License Agreement". Then you need to click on the link that has been highlighted in the below image

## JAVA TUTORIAL



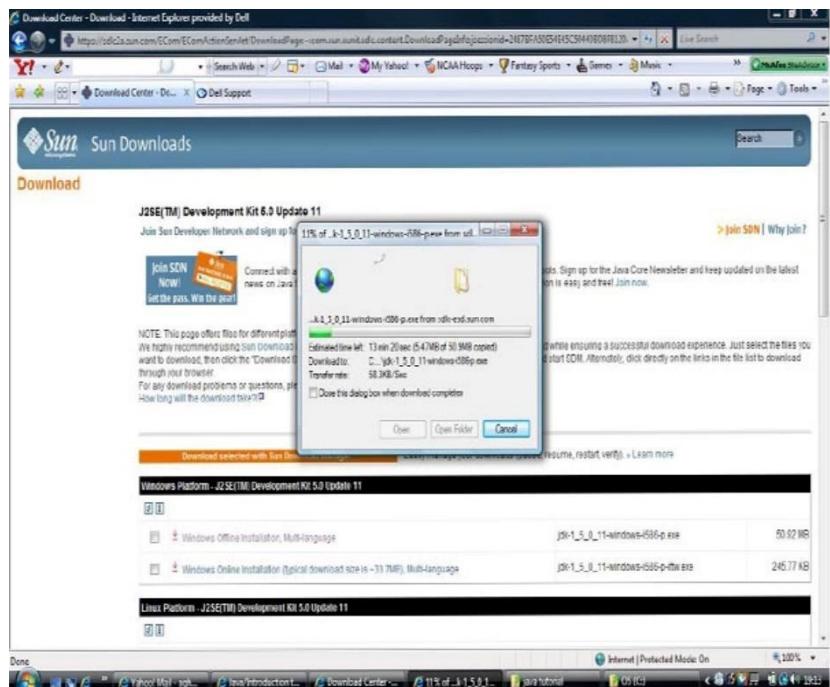
5. Then it will ask you to save the file. Choose any directory location on your computer and click on "Save" button.

## JAVA TUTORIAL



6. It will immediately start downloading.

## JAVA TUTORIAL



7. Once it is downloaded on your system you have to install it on your PC/laptop. Just follow the instructions given during the installation. Believe me it is not difficult to install JDK.

8. After installing JDK 1.5 you will be able to execute any java file.

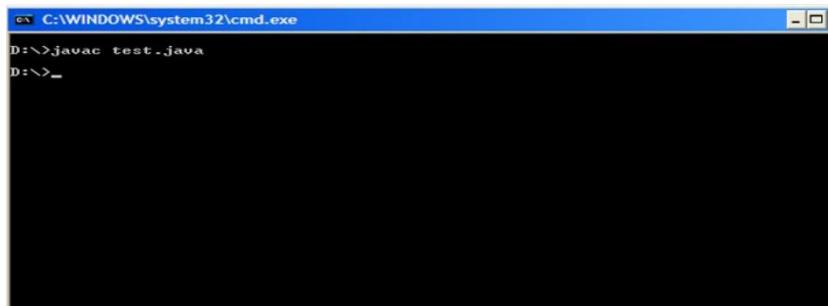
## JAVA TUTORIAL

9. To check whether JDK 1.5 has been installed properly on your PC/laptop create a file called test.java and add the below code in it.

```
public class test {  
  
    public static void main(String args[]) {  
  
        System.out.println("Java is working properly on your  
machine");  
  
    }  
  
}
```

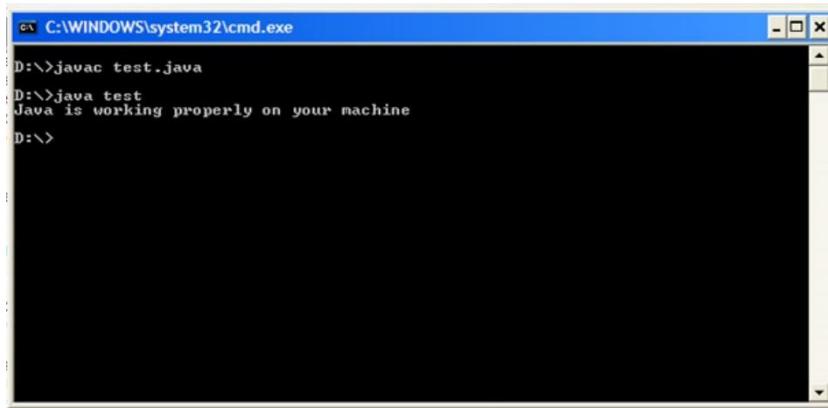
10. Open a cmd window and go to the directory location where you saved test.java. Execute the command **javac test.java**. javac is a tool used to compile a java file. Once it compiles a java file successfully it will automatically create a file called test.class in the same directory.

## JAVA TUTORIAL



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the command 'javac test.java' being typed at the prompt 'D:\>'. The command has been partially entered.

11. Run the command **java test**.



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the commands 'javac test.java' and 'java test' being typed at the prompt 'D:\>'. After 'java test', the output 'Java is working properly on your machine' is displayed. The command prompt then returns to 'D:\>'.

**Congratulations!!!**, You have successfully installed java on your system.

## JAVA TUTORIAL

# Applications of JAVA

- Mobile Applications
- Desktop GUI Applications
- Web-based Applications
- Enterprise Applications
- Scientific Applications
- Gaming Applications
- Big Data technologies
- Business Applications
- Distributed Applications
- Cloud-based Applications

# Data types and Type casting

Java is termed as strongly typed Programming which means that every variable which is defined in any line code of java programming has to be properly well defined and assigned a data type and this again makes it possible for java to maintain such complex robustness and security to the code and the data.

If we broadly categorize and classify these data types, we can define them as Eight primitive types of data types available to us in Java.

1. Four of them are integer types
2. Two are floating-point number types

## JAVA TUTORIAL

3. One is the character type char, used for code units in the Unicode encoding scheme
4. One is a Boolean type for truth values

In most situations, the int type is the most practical. If you want to represent the number of inhabitants of our planet, you'll need to resort to a long. The byte and short types are mainly intended for specialized applications, such as low-level file handling, or for large arrays when storage space is at a premium.

Let's take these four categories in brief for our better understanding

1. **Integers** can be defined as byte, short, int, and long, which are for whole- Valued signed numbers.

The int type is the most practical. If you want to represent the number of humans on earth you'll need to resort to a long. The byte and short types are mainly intended for specialized applications.

**JAVA TUTORIAL**

<i>Java Integer Types</i>		
Type	Storage Requirement	Range (Inclusive)
Int	4 bytes	-2,147,483,648 to 2,147,483, 647 (just over 2 billion)
Short	2 bytes	-32,768 to 32,767
Long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Byte	1 byte	-128 to 127

2. **Floating-point** numbers basically is used when we have the situation of getting our result or output in the form of decimals and not whole numbers as mentioned in case of Integers Data Types. This group thus includes float and double.

<i>Java Floating Types</i>		
Type	Storage Requirement	Range (Inclusive)
double	64 bits	4.9e324 to 1.8e+308
float	32 bits	1.4e045 to 3.4e+038

3. **Characters** help defining the char, which represents symbols in a character Set, like letters and numbers.

## JAVA TUTORIAL

Character Escape Sequences	
Escape Sequence	Description
\uxxxx	Hexadecimal UNICODE character (xxxx)
\'	Single quote
\"	Double quote
\\\	Backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Tab
\b	Backspace

4. **Boolean** data type is used to mention the variables which will only have the possibility of containing values either as True or False.

## Variables and Arrays

### ***Variable***

The variable is the basic unit of storage in a Java program. You declare a variable by placing the type first, followed by the name of the variable. Here are some examples:

```
double salary;
```

```
double _salary;
```

```
int holidays;
```

**JAVA TUTORIAL**

```
int $holidays;  
long distanceFromMoonToEarth;  
boolean done;
```

Notice the semicolon at the end of each declaration. The semicolon is necessary because a declaration is a complete Java statement. In addition, all variables have a scope, which defines their visibility, and a lifetime. General form of declaring variables:

**type identifier [= value][, identifier [= value] ...] ;**

Where type identifier could be byte, int, long, boolean, char, short, float, double. Variable name cannot start with numeric character or special characters except "\$" and "\_". For valid example please refer above variable names. Let us also see some invalid variable names:

```
double 1salary; // invalid variable name  
double %salary; // invalid variable name  
int #holidays; // invalid variable name
```

After you declare a variable, you must explicitly initialize it by means of an assignment statement—you can never use the values of uninitialized variables. For example, the Java compiler flags the following sequence of statements as an error.

**JAVA TUTORIAL**

```
byte z = 22;
```

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
float a = 10.0f
```

```
float b = 3.4f
```

```
float i = a / b;
```

***Scope and Lifetime of Variables***

Lifetime of a variable, that is, the time a variable is accessible during the execution, is determined by the context in which it is declared. We distinguish between lifetime of variables in three contexts:

**Instance variables** - members of a class and created for each object of the class. Every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the state of the object. Instance variables exist as long as the object they belong to exists.

**Static variables** - also members of a class, but not created for any object of the class and, therefore, belong only to the class. They are not associated with any object. They are created when

## JAVA TUTORIAL

the class is loaded at runtime, and exist as long as the class exists. Every object derived from the class will share static variables. In a multithreaded application static variable are discouraged because of their feature.

**Local variable** - declared in methods and in blocks and created for each execution of the method or block. After the execution of the method or block completes local variables are no longer accessible. Local variable should be initialized within the methods and blocks where they are created. Not doing so will cause java compilation error.

### **Arrays**

An array is a data structure that defines an indexed collection of a fixed number of homogeneous data elements. In java arrays are objects that store multiple variables of the same type. Arrays can hold either primitives or object references, but the array itself will always be an object on the heap, even if the array is declared to hold primitive elements. A position in the array is indicated by a non-negative integer value called index. An element at a given position in the array is accessed using the index. The size of an array is fixed and cannot be increased to accommodate more elements.

#### ***Declaring an Array***

**JAVA TUTORIAL**

Arrays are declared by stating the type of elements the array will hold, which can be an object or primitive followed by square brackets to the left or right of the identifier.

<element type>[] <array name>;

or

<element type> <array name>[];

Arrays can be single dimensional, two dimensional or multidimensional. Declaring a single dimensional array is very simple. The most common way to construct an array is to use the new keyword that allocates memory to any object. Remember arrays are treated as objects in Java. The following is an example of constructing a single dimensional array of type long:

```
long[] empSalaries = new long[10];
```

As you can see that the above statement will allocate memory to the array variable empSalaries and the size of the array would be 10. In java index begin from 0. That means in order to access the first element of an array you need to do something like

```
empSalaries[0];
```

```
public SingleDimensionalArray {
```

## JAVA TUTORIAL

```
long[] empSalaries = new long[] {10000, 20000, 30000,  
40000};  
  
public static void main(String[] args) {  
  
    System.out.println("Value of the first element in the array:  
" + empSalaries[0];  
  
}  
  
}
```

An array can be initialized in many ways:

1) long[] empSalaries = {10000L, 20000L, 30000L, 40000L};

// L means long value. It is required to distinguish between  
integer value and

// long value. By default it is integer always.

2) long[] empSalaries = new long[] {10000L, 20000L,  
30000L, 40000L};

3) long[] empSalaries = new long[4];

empSalaries[0] = 10000L;

empSalaries[1] = 20000L;

empSalaries[2] = 30000L;

## JAVA TUTORIAL

```
empSalaries[4] = 40000L;
```

Let us check out some invalid array declarations

```
String names[] = new String[]; // We have to mention the size  
of
```

```
// the array when using new operator.
```

```
int ars[] = new int{1,2,3,4,5}; // square([]) brackets are  
missing.
```

```
int ars[] = new int[5]{1,2,3,4,5}; // size of the array should not  
be mentioned
```

```
// when declaring an array like this.
```

```
int ars[4]; // it is invalid because memory can be allocated  
only using new operator.
```

```
int ars[];
```

```
ars = {1,2,3,4,5};
```

### ***Multidimensional Array***

In Java multidimensional arrays are actually arrays of arrays. These as you might expect, look and act like regular multidimensional arrays. Multidimensional arrays use more than one index to access array elements. They are used for

## JAVA TUTORIAL

tables and other more complex arrangements. Let us take an example of two-dimensional array which is a form of multidimensional array.

```
int matrix = new int[3][2];
```

This very much resembles to a 3 x 2 matrix. Where 3 represents rows and 2 represents columns.

Valid array declarations

```
Integer ars[][] = new Integer[4][3]; // Integer is a wrapper class for int.
```

```
int ars[][] = new int[][]{{1,2,3}, {2,4,6}, {3,4,7}, {5,6,7}};
```

It will be like a 4 x 3 matrix

—	—	
	1 2 3	
	2 4 6	
	3 4 7	
	5 6 7	
+-	-+	

***Iterate through a multidimensional array***

JAVA TUTORIAL

```
public class MultiDimensionalArrayExample {  
  
    public static void main(String[] args) {  
  
        int ars[][] = new int[][]{{1,2,3}, {2,4,6}, {3,4,7}, {5,6,7}};  
  
        for(int i = 0; i < 4; i++) {  
            for(int j = 0; j < 3; j++) {  
                System.out.println(ars[i][j]);  
            }  
        }  
    }  
}
```

## ***2 Operators and Statements***

---

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: assignment, arithmetic, relational, and logical. Java also defines some additional operators that handle certain special situations.

### **Operators**

#### ***Assignment Operators***

The assignment statements have the following syntax:

<variable> = <expression>

The destination variable and the source expression must be type compatible. The destination variable must also have been declared. Since variable can store either primitive data values or object references, <expression> evaluates to either a primitive data value or an object reference. When assigning a value to a primitive, size matters. Be sure you know when implicit casting will occur, when explicit casting is necessary, and when truncation might occur.

Assigning primitive value

```
int a, b;  
a = 2; // 2 is assigned to variable a  
b = 5; // 5 is assigned to variable b
```

## JAVA TUTORIAL

### *Assigning references*

```
Home home1 = new Home(); // new object created
```

```
Home home2 = home1; // assigning the reference of home1 in  
home2
```

In the above example home1 and home2 points to the same reference. Any change made to attribute in home2, same change will be reflected in the attribute of home1 also. But interesting thing is that if you assign null value to home2 it will be applied only on home2 and not on home1. home1 will still be holding the reference and home2 will be nullified.

### **Compound Assignment Operators**

`+=`, `-=`, `*=`, and `/=`

### *Examples*

```
int i;
```

```
i += 5; // it is equal to i = i + 5
```

```
int a;
```

```
a -= 5; // it is equal to i = i - 5
```

```
int b;
```

## JAVA TUTORIAL

```
b *= 5 // it is equal to i = i * 5
```

```
int c;
```

```
c /= 5 // it is equal to i = i / 5
```

### ***Arithmetic Operators***

The arithmetic operators are used to construct mathematical expressions as in algebra. Their operands are of numeric type.

+ addition        //  $i = i + 1$

- subtraction      //  $i = i - 1$

\* multiplication    //  $i = i * 1$

/ division        //  $i = i / 1$ . This operator only returns the quotient.

% remainder operator //  $i = i \% 1$ . This operator returns the remainder

                        // value after division.

++ increment operator //  $i++$  . It increments the value by one.

-- decrement operator //  $i--$  . It decrements the value by one.

If you have any doubt about the usage of division operator and remainder operator then please try the below code.

#### Code for Division operator:

```
public DivisionExample {
```

## JAVA TUTORIAL

```
public static void main(String args[]) {  
  
    int a = 6;  
  
    System.out.println(a / 4);  
  
}  
}
```

### Code for remainder operator

```
public RemainderOpertor {
```

```
public static void main(String args[]) {  
  
    int a = 6;  
  
    System.out.println(a % 4);  
  
}  
}
```

## JAVA TUTORIAL

In Java, + operator is also used for String concatenation. Like an example given below

```
public StringConcatenation {  
  
    public static void main(String[] args) {  
  
        String firstName = "FirstName";  
        String lastName = "LastName";  
        String name = firstName + " " + lastName;  
  
        System.out.println("Name is " + name);  
    }  
}
```

### *Relational Operators*

Java has six relational operators

< Less than sign

<= Less than or equal to sign

## JAVA TUTORIAL

> Greater than sign

=> Greater than or equal to sign

== Equal to sign

!= Not equal to sign

Less than sign is used to check whether the value of the first variable is less than the value of the second variable.

```
public LessThanExample {
```

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

```
        int a = 5;
```

```
        int b = 10;
```

```
        if(a < b) {
```

## JAVA TUTORIAL

```
System.out.println("a is less than b");

}

}

}
```

Less than or equal to sign is used to check whether the value of the first variable is less than or equal to the value of the second variable.

```
public LessThanEqualExample {
```

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

```
    int a = 10;
```

```
    int b = 10;
```

```
    if(a <= b) {
```

```
        System.out.println("a is less than or equal to b");
```

```
    }
```

```
}
```

**JAVA TUTORIAL**

```
}
```

Greater than or equal to sign is used to check whether the value of the first variable is greater than or equal to the value of the second variable.

```
public GreaterThanEqualExample {
```

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

```
        int a = 10;
```

```
        int b = 10;
```

```
        if(a => b) {
```

```
            System.out.println("a is greater than or equal to b");
```

```
        }
```

```
    }
```

```
}
```

Equal to sign is used to check whether the value of the first variable is equal to the value of the second variable.

## JAVA TUTORIAL

```
public EqualToExample {  
  
    public static void main(String args[]) {  
  
        int a = 10;  
        int b = 10;  
  
        if(a == b) {  
  
            System.out.println("a is equal to b");  
        }  
    }  
}
```

Not equal to sign is used to check whether the value of the first variable is not equal to the value of the second variable.

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

## JAVA TUTORIAL

```
int a = 10;  
int b = 5;  
  
if(a != b) {  
    System.out.println("a is not equal to b");  
}  
}  
}
```

### ***Logical Operators***

These logical operators work only on boolean operands. Their return values are always boolean.

? : Ternary if-then-else

& Logical AND

&& Short-circuit AND

| Logical OR

## JAVA TUTORIAL

|| Short-circuit OR

^ Logical XOR

== Equal to

!= Not equal to

! Logical unary NOT

Now we will see the usage of these operators.

? : *Ternary if-then-else*

```
public class TernaryOperatorExample {
```

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

```
        char ans = 'y';
```

**JAVA TUTORIAL**

```
// (?) condition is checked  
// (:) value is returned based on the condition.  
If  
    // it is true then the first expression is  
    returned  
        // otherwise the second one.  
  
String value = ans == 'y' ? "IT'S YES" : "OH  
NO!!!";  
  
System.out.println(value);  
  
}  
}
```

**& Logical AND**

```
public class ANDOperatorExample {
```

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

```
    char ans = 'y';
```

```
    int count = 1;
```

## JAVA TUTORIAL

```
if(ans == 'y' & count == 0) {  
    System.out.println("Count is  
Zero.");  
}  
  
if(ans == 'y' & count == 1) {  
    System.out.println("Count is  
One.");  
}  
  
if(ans == 'y' & count == 2) {  
    System.out.println("Count is  
Two.");  
}  
}
```

### && Short-circuit AND

```
public class ShortCircuitANDOperatorExample {
```

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

**JAVA TUTORIAL**

```
int hour = 9;  
int minute = 30;  
  
// short circuit AND operator is differnt from  
// normal AND operator. If happens like this  
that  
  
// the when first condition is checked and if it  
// is true then only it checks for the second  
condition  
  
// that is on the other side of the &&.  
  
if (hour == getHour() && minute ==  
getMinute()) {  
  
    System.out.println("It is 10:30");  
}  
  
}  
  
  
public static int getHour() {  
    System.out.println("Return Hour");
```

## JAVA TUTORIAL

```
        return 10;  
    }  
  
public static int getMinute() {  
    System.out.println("Return Minute");  
    return 30;  
}  
}
```

### *Logical OR*

```
public class ORExample {  
  
public static void main(String[] args) {  
  
    int x = 10;  
    if(x == 5 | x ==10) {  
        System.out.println("Value of x  
can be divided by 10");  
    }  
}
```

**JAVA TUTORIAL**{ }  
}***Short-circuit OR*****public class** ShortCircuitORExample {**public static void** main(String[] args) {    **int** hour = 10;    **int** minute = 30;        *// if the first condition is true then it will not*        *// check for the second condition. But if the*  
*first*        *// condition is false then it will check for*  
*second*        *// condition. In case of OR either one of the*  
*conditions*        *// should be true.*    **if**(hour == getHour() || minute ==  
getMinute()) {

## JAVA TUTORIAL

```
System.out.println("Correct time  
of display this message");
```

```
}
```

```
public static int getHour() {
```

```
System.out.println("returning hour");
```

```
return 10;
```

```
}
```

```
public static int getMinute() {
```

```
System.out.println("returning minute");
```

```
return 30;
```

```
}
```

```
}
```

### *Logical XOR*

```
public class XORExample {
```

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

## JAVA TUTORIAL

```
int a = 1;
```

```
int b = 2;
```

```
int c = 3;
```

```
int d = 4;
```

// XOR will return true only if the conditions

// are mutually exclusive.

System.out.println(a == 1 ^ b == 2); // both

the conditions are true

System.out.println(c == 4 ^ d == 3); // both

the conditions are false

System.out.println(c == 4 ^ d == 4); // only

second condition is true

```
}
```

```
}
```

***Equal to***

```
public class EqualToExample {
```

## JAVA TUTORIAL

```
public static void main(String[] args) {  
  
    char ans = 'y';  
  
    if(ans == 'y') {  
        System.out.println("It is YES.");  
    }  
}  
}
```

*Not equal to*

```
public class NotEqualToExample {  
  
    public static void main(String[] args) {  
  
        int value = 4;  
        if (value != 10) {  
            System.out.println("Value is not  
equal to 10");  
        }  
    }  
}
```

**JAVA TUTORIAL**

```
}
```

```
}
```

***Logical unary NOT***

```
public class UnaryNotExample {
```

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

```
        boolean flag = false;
```

*// Here the value of the flag is false. But if  
block*

*// is only executed when the condition is true.*

*So*

*// in cases we can use Unary Not (!). This  
negates the*

*// boolean value of the flag. If the value of  
flag is*

*// false and used with Unary Not then it will  
return*

*// true and vice-versa.*

```
    if(!flag) {
```

## JAVA TUTORIAL

```
System.out.println("Hello  
world.");  
}  
}  
}
```

## Statements

The control statements provided in Java are basically used to cause the proper flow of execution in the advanced and branched Java programs. These control statements can be categorized into the following:

### *Selection Statements*

Selection statements, as self-explanatory helps the programmer to choose different paths of execution based upon the outcome of an expression or the state of a variable.

```
if(isOpen) { //isOpen is a boolean variable
```

```
    System.out.println("The Door is OPEN");
```

```
} else {
```

## JAVA TUTORIAL

```
System.out.println("The Door is CLOSE);
```

```
}
```

```
<>
```

### ***Iteration Statements***

Iteration statements as the term states, is the repetition of the block of code in a program.

```
int i = 1;
```

```
while(i < 5) {
```

```
    System.out.println("The current value of variable i is " + i);
```

```
    i = i + 1;
```

```
}
```

### ***Jump Statements***

Jump statements allow your program to execute in a nonlinear fashion.

## JAVA TUTORIAL

```
// break is a jump statement

int count = 5;

while(true) {
    if(count == 5) {

        System.out.println("We will use break statement " +
                           "to jump out of the while block");

        break;
    }

    // code for some process.
}
```

## Control Structures

Control structures are programming blocks that can change the path we take through those instructions.

There are three kinds of control structures:

- Conditional Branches, which we use **for choosing between two or more paths**. There are three types in Java: *if/else/else if, ternary operator and switch*.

## JAVA TUTORIAL

- Loops that are used to **iterate through multiple values/objects and repeatedly run specific code blocks.** The basic loop types in Java are *for*, *while* and *do while*.
- Branching Statements, which are used to **alter the flow of control in loops.** There are two types in Java: *break* and *continue*.

These are used to choose the path for execution. There are some types of control statements:

1. if statement: It is a simple decision-making statement. It is used to decide whether the statement or block of statements will be executed or not. Block of statements will be executed if the given condition is true otherwise the block of the statement will be skipped.

```
if(condition) {  
    // If condition is true then block of statements will be executed  
}
```

1. nested if statement: Nested if statements mean an if statement inside an if statement. The inner block of **if statement** will be executed only if the outer block condition is true.
2. if-else statement: In **if statement** the block of statements will be executed if the given condition is true otherwise block of the statement will be skipped. But we also want to execute the same block of code if the condition is false.

## JAVA TUTORIAL

Here we come on the **if-else statement**. An **if-else statement**, there are two blocks one is **if block** and another is **else block**. If a certain condition is true, then **if block** will be executed otherwise **else block** will be executed.

```
if (condition) {  
    // It will be print if block condition is true.  
}  
  
else  
{  
    // It will be print if block condition is false.  
}
```

1. **if-else if statement/ ladder if statements:** If we want to execute the different codes based on different conditions then we can use **if-else-if**. It is also known as **if-else if ladder**. This statement is always be executed from the top down. During the execution of conditions if any condition founds true, then the statement associated with that if it is executed, and the rest of the code will be skipped. If none of the conditions is true, then the final else statement will be executed.

```
if(condition) {  
    // If condition is true then this block of statements will be executed
```

## JAVA TUTORIAL

```
}
```

```
else if(condition) {
```

```
// If condition is true then this block of statements will be executed
```

```
} ...
```

```
Else
```

```
{
```

```
// If none of condition is true, then this block of statements will be
```

```
executed
```

```
}
```

1. **Switch statement:** The switch statement is like the if-else-if ladder statement. To reduce the code complexity of the if-else-if ladder switch statement comes. In a switch, the statement executes one statement from multiple statements based on condition. In the switch statements, we have a number of choices and we can perform a different task for each choice.

```
switch(variable/expression)
```

```
{
```

```
case value1 : // code inside the case value1
```

```
break; //
```

```
optional case value2 : // code inside the case value2
```

```
break; //
```

**SHOIAB MOHAMMAD**

**9840195749**

**JAVA TUTORIAL**

optional . . . default : // code inside the default case .

}

## JAVA TUTORIAL

These are used to iterate the instruction for multiple times.

1. **for loop:** If a user wants to execute the block of code several times. It means the number of iterations is fixed. JAVA provides a concise way of writing the loop structure.

```
for(initialization; condition; increment/decrement)  
{  
    // Body of for loop  
}
```

1. **while loop:** A **while loop** allows code to be executed repeatedly until a condition is satisfied. If the number of iterations is not fixed, it is recommended to use a **while loop**. It is also known as an **entry control loop**.

```
while(condition)  
{  
    // Body of while loop  
}
```

1. **do-while loop:** It is like while loop with the only difference that it checks for the condition after the execution of the body of the loop. It is also known as **Exit Control Loop**.

```
do
```

## JAVA TUTORIAL

```
{  
// Body of loop  
} While(condition);
```

1. **Enhanced for loop:** This is mainly used to traverse collection of elements including arrays.

### Syntax

```
for(declaration : expression) { // Statements }
```

- **Declaration** – The newly declared block variable, is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression** – This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

These are used to **alter the flow of control in loops.**

1. **break:** If a **break statement** is encountered inside a loop, the loop is immediately terminated, and the control of the program goes to the next statement following the loop. It is used along with **if statement in loops.** If the condition is true, then the **break statement** terminates the loop immediately.

```
break;
```

## JAVA TUTORIAL

1. continue: The continue statement is used in a loop control structure. If you want to skip some code and need to jump to the next iteration of the loop immediately. Then you can use a **continue statement**. It can be used for loop or while loop.

```
continue;
```

# **3 OOPs**

---

## ***Object oriented Programming***

Java is totally an object-oriented programming language as it works totally on the concepts of Object-oriented Programming concepts. We will try and understand the basic concept of Object-Oriented Programming language and the concept of object in the section below.

Object oriented programming (called **OOPS** in short) is the concept, which got its way after the procedural programming languages which was developed in 1970's. The programming language which existed before was more of process oriented and this gave the concept of small entities called objects which could be made and reused as and when the need arises.

OOPs hails from the idea of Objects. All the real-life entities are basically nothing but objects. In any program every code will have an object having few characteristics features called the properties of that particular object or entity and then it will always have few actions to be performed over those entities or objects termed as Methods or Functions. To work with OOP, you should be able to identify three key characteristics of objects:

- The behavior of object — what can you do with this object, or what methods can you apply to it?
- The state of the object— how does the object react when you apply those methods?
- The identity of the object— how is the object distinguished from others that may have the same behavior and state?

## JAVA TUTORIAL

This would have surely given an overview as to what is OOPs, why and how it has resulted in the beginning of a new era in programming language but to have a proper understanding on OOPs, it is very important to make yourself familiar with one more term Class.

### **Class**

The objects actually originate from the class or to define in an easy way the class is the template from which the object is derived. Whenever we create an object from the class it is called the instance of that particular class. All code that you write in Java is inside a class only. For example, there is a class called Animal and it has an instance Dog as an object to the class Animal and hence called the instance of it.

### **Principles of OOPs**

All object programming languages provide us with few basic mechanisms which can be termed as Basic principles of OOPs and they are

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**

We will discuss each one of them in detail in the section below as they hold the key to successful implementation to the object-oriented Model approach.

## JAVA TUTORIAL

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs.

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around “what is happening” and others are written around “who is being affected.”

The first way is called the process-oriented model. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success.

To manage increasing complexity, the second approach, called object-oriented programming, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

**Simula** is considered the first object-oriented programming language. The programming paradigm where everything is

## JAVA TUTORIAL

represented as an object is known as a truly object-oriented programming language.

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

### *Abstraction*

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on.

## JAVA TUTORIAL

The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior

### *Encapsulation*

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

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.

## JAVA TUTORIAL

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 IDEs are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

In Java, the basis of encapsulation is the class. A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class.

Thus, a class is a logical construct; an object has physical reality. The data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods.

### *Inheritance*

**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). This is important because it supports the concept of hierarchical classification.

## JAVA TUTORIAL

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. A general class is one that defines traits common to a set of related objects, i.e., objects with common attributes and behaviours.

Inheritance leads to the definition of generalized classes that are at the top of an inheritance hierarchy. Inheritance is thus the implementation of generalization. Inheritance, in an object-oriented language like Java makes the data and methods of a **superclass** available to its **subclass**. Inheritance has many advantages, the most important of them being the reusability of code. Once a class has been created, it can be used to create new subclasses.

### ***Polymorphism***

Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler’s job

## JAVA TUTORIAL

to select the specific action (that is, method) as it applies to each situation.

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 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.

## Classes, Objects and Methods

**Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class. In Java everything is **encapsulated under classes**. Class is the core of Java language. It can be defined as a **template** that describe the behaviors and states of a particular entity.

A class defines new data type. Once defined this new type can be used to create object of that type.

An object has three characteristics:

## JAVA TUTORIAL

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

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

### 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*.

**Object** is a superclass of all other classes; i.e., Java's own classes, as well as user-defined classes. This means that a reference variable of type **Object** can refer to an object of any other class. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The **new** operator dynamically allocates (that is,

## JAVA TUTORIAL

allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.

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**

A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex.

In general, class declarations can include these components, in order:

1. **Modifiers:** A class can be public or has default access (Refer [this](#) for details).
2. **class keyword:** class keyword is used to create a class.

## JAVA TUTORIAL

3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body surrounded by braces, { }.

Example:

```
class classname {  
    type instance-variable1;  
    type instance-variable2; // ...  
    type instance-variableN;  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    } // ...
```

## JAVA TUTORIAL

```
type methodnameN(parameter-list) {  
    // body of method  
}  
}
```

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

A method must be declared within a class. It is defined with the name of the method, followed by parentheses (). Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions

This is the general form of a method:

```
type name(parameter-list) {  
    // body of method  
}
```

Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void. The name of the method is specified by name. This can be any legal

## JAVA TUTORIAL

identifier other than those already used by other items within the current scope. The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than void return a value to

the calling routine using the following form of the return statement: return value; Here, value is the value returned.

For using a method, it should be called. There are two ways in which a method is called i.e., method returns a value or returning nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when –

- The return statement is executed.
- It reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Let's consider an example –

```
System.out.println("This is JAVA!");
```

The method returning value can be understood by the following example –

```
int result = sum(6, 9);
```

## JAVA TUTORIAL

Most of the time, you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

## Modifiers

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.

Modifiers are keywords that you add to those definitions to change their meanings. Java language has a wide variety of modifiers, including the following –

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.

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved.

## JAVA TUTORIAL

### *Access Modifiers*

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are –

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

When a member of a class is modified by public, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. Now you can understand why main( ) has always been preceded by the public modifier. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

The private access modifier is accessible only within the class. If you make any class constructor private, you cannot create the instance of that class from outside the class. 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. The **protected access**

## JAVA TUTORIAL

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

### *Non access modifiers*

Java provides a number of non-access modifiers to achieve many other functionalities.

- The *static* modifier for creating class methods and variables.
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

### The Static Modifier

#### Static Variables

The *static* keyword is used to create variables that will exist independently of any instances created for the class. Only one copy

## JAVA TUTORIAL

of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

### Static Methods

The static keyword is used to create methods that will exist independently of any instances created for the class.

Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

### The Final Modifier

#### Final Variables

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

#### Final Methods

## JAVA TUTORIAL

A final method cannot be overridden by any subclasses. As mentioned previously, the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

### Final Classes

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

### The abstract Modifier

#### Abstract Class

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

#### Abstract Methods

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass. Abstract methods can never be final or strict.

## JAVA TUTORIAL

Any class that extends an abstract class must implement all the abstract methods of the super class, unless the subclass is also an abstract class.

If a class contains one or more abstract methods, then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon.

Example: public abstract sample();

### The Synchronized Modifier

The synchronized keyword used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

### The Volatile Modifier

The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

## Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical

## JAVA TUTORIAL

classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the members defined by the superclass and adds its own, unique elements.

To define a new class from an already existing class, you incorporate the definition of an already existing class into the class to be defined by using the ***extends*** keyword.

### ***Using super()***

A subclass should not have direct access to the data members of its superclass

It is an important principle of class design that every class should keep the details of its implementation to itself

One aspect is to keep its data **private** to itself.

This should hold true even in the case of a superclass-subclass relationship in a class hierarchy

The subclass should not have access to the implementation details of its superclass

Therefore, data members should be private in a superclass

Otherwise, encapsulation can be broken at will by just creating a dummy subclass and accessing all the superclass data members

## JAVA TUTORIAL

The aforesaid conclusion apparently seems to give rise to another problem

Consider a situation where you straight away and independently create an object of the subclass without first creating an object of the superclass

The key to this problem is the **super** keyword. **super** has two uses.

The first is the call to the superclass' constructor from the subclass constructor

The second usage is to access a member of the superclass that has been overridden by a subclass

A subclass constructor can call a constructor defined in its immediate superclass by employing the following form of **super**:

```
super(parameter-list);
```

## Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that

## JAVA TUTORIAL

the code creating an instance will have a fully initialized, usable object immediately.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

### *Parameterized Constructors*

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

## Destructors

In Java, when we create an object of the class it occupies some space in the memory (heap). If we do not delete these objects, it remains in the memory and occupies unnecessary space that is not upright from the aspect of programming. To resolve this problem, we use the **destructor**. In this section, we will discuss the alternate

## JAVA TUTORIAL

option to the **destructor in Java**. Also, we will also learn how to use the **finalize()** method as a destructor.

The **destructor** is the opposite of the constructor. The constructor is used to initialize objects while the destructor is used to delete or destroy the object that releases the resource occupied by the object. Java does not support this idea or provide for destructors. The **finalize()** method only approximates the function of a destructor. As you get more experienced with Java, you will see that the need for destructor functions is minimal because of Java's garbage collection subsystem.

For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

The **finalize( )** method has this general form:

```
protected void finalize( ) {
```

**JAVA TUTORIAL**

```
// finalization code here }
```

Here, the keyword `protected` is a specifier that limits access to `finalize()`.

## Abstractions

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

It makes little sense for a partially defined superclass to be instantiated, for, there are going to be some methods with no implementations. The importance of such classes cannot be discounted because they define a generalized form (possibly some generalized methods with no implementations) that will be shared by all of its subclasses, leaving it to each subclass to provide for its own specific implementations of such methods.

Java's solution to this problem is the abstract method.

You can require that certain methods be overridden by subclasses by specifying the `abstract` type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a

## JAVA TUTORIAL

subclass must override them—it cannot simply use the version defined in the superclass.

To declare an abstract method, use this general form:

**abstract type name(parameter-list);**

As you can see, no method body is present. Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the `abstract` keyword in front of the `class` keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the `new` operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

The key, therefore, is to have a mechanism to ensure that a subclass does, indeed, override, all necessary methods. This mechanism comes in the shape of the abstract method.

Abstract methods in a superclass **must** be overridden by its subclasses as the subclasses cannot use the abstract methods that they inherit

These methods are sometimes referred to as **subclasses' responsibility** as they have no implementation specified in the superclass

**JAVA TUTORIAL**

## Inner classes

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used. The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

```
// Demonstrate an inner class.
```

```
class Outer {  
    int outer_x = 100;  
    void test() {
```

## JAVA TUTORIAL

```
Inner inner = new Inner();  
  
inner.display();  
} // this is an inner class  
  
class Inner {  
  
void display() {  
  
System.out.println("display: outer_x = " + outer_x);  
  
}  
  
}  
  
}  
  
class InnerClassDemo {  
  
public static void main(String args[]) {  
  
Outer outer = new Outer(); outer.test();  
  
}  
  
}
```

Output: **display: outer\_x = 100**

## Packages

### Need for packages

Till now, you did not use any package since all your classes were stored in the **default** package. Imagine a situation where all the

## JAVA TUTORIAL

classes are stored in one package. It would lead to tremendous confusion as it may lead to classes with similar names that is not allowed by the language.

**This scenario is also referred to as a namespace collision.**

The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the package statement:

**package pkg;**

Visibility control refers to what classes can be accessed from another package, and what class members can be accessed from other classes.

As you know, **classes** are at the core of Java language. There are numerous built-in classes that provide rich functionality to the language. These classes are also organized into packages depending upon their functionality leading to functionally cohesive packages.

## JAVA TUTORIAL

For example :

*java.applet* contains classes related to applets.

*java.net* contains classes related to networking

*java.util* contains classes related to utilities like-date, calendar etc.

*java.awt* contains classes related to GUI – Graphical User Interface.

**You can also have a package within a package.**

The first statement defines a package called **MyPack**. The keyword package followed by a package-name creates a package by the package-name. All the classes in your program are stored in the package **MyPack**.

Till now, you did not write any such statement, so all your classes were stored in the default package.

This is okay for sample programs, but for real applications you should keep your classes in packages. You can store one package into another. This is how numerous classes in the Java language are arranged. For example:

1. java.awt
2. java.awt.event

Hence, we can have packages stored in a hierarchy. Now, we can redefine **package as a container for classes, interfaces and packages**.

## JAVA TUTORIAL

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

Keep all the **.java** files in the directory **MyPack**. In real applications, the same package may be used in many files. This indicates that all the classes created in those files are stored in same package. Hence, you should save all those **.class** files in the same directory. If there is a hierarchy of packages, you should also have a hierarchy of directories.

For example, `java.awt.event` will be stored in the directory

`java\awt\event` – in Windows

`java/awt/event` – in UNIX

**If you want to rename the package you should rename the directory also.**

## JAVA TUTORIAL

# ClassPath

Hence, you did not face any problem in executing your programs.

What is **CLASSPATH**?

**CLASSPATH is an environment variable that tells the Java runtime system where the classes are present**

When a package is not created, all classes are stored in the default package. **The default package is stored in the current directory. The current directory is the default directory for CLASSPATH.**

How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable. Third, you can use the -classpath option with java and javac to specify the path to your classes.

For example, consider the following package specification:  
package MyPack in order for a program to find MyPack, one of three things must be true. Either the program can be executed from a directory immediately above MyPack, or the CLASSPATH must be set to include the path to MyPack, or the -classpath option must specify the path to MyPack when the program is run via java.

**JAVA TUTORIAL**

When the second two options are used, the class path must not include MyPack, itself. It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is C:\MyPrograms\Java\MyPack then the class path to MyPack is C:\MyPrograms\Java The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the .class files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

## Importing Packages

Java has used the package mechanism extensively to organize classes with similar functionality in one package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.

The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing. In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement: import pkg1 [.pkg2].(classname | \*);

## JAVA TUTORIAL

If you want to use these classes in your applications, you can do so by including the following statement at the beginning of your program:

*import packagename.classname;*

*If the packages are nested you should specify the hierarchy.*

*import package1.package2.classname;*

A package called **java.lang** contains the language-related classes. This package is implicitly imported for you.

When you specify a star(\*) in the import statement, all the classes, interfaces in the package are imported. But if we need only one or two of them, it is advisable to import them specifically. This approach would reduce compile time.

## Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.

## JAVA TUTORIAL

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses.

Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

An interface consists of abstract methods.

Any class that implements an interface should provide implementation for the methods given in the interface. Each class, implementing an interface chooses to keep its own implementation of the same behaviour defined by an interface specific to its needs. If a class fails to do so, it must be declared abstract.

Defining an interface has the advantage that an interface definition stands apart from any class or class hierarchy

Moreover, it consists of a specification of behaviour/s in the form of abstract methods

## JAVA TUTORIAL

This makes it possible for any number of independent classes to implement the interface

Thus, an interface is a means of specifying a consistent specification, the implementation of which can be different across many independent and unrelated classes to suit the respective needs of such classes

Java does not support multiple inheritance

This is a constraint in class design, as a class cannot achieve the functionality of two or more classes at a time

Interfaces help us make up for this loss. A class can implement more than one interface at a time

Thus, interfaces enable you to create richer classes and at the same time the classes need not be related

The utility of abstract methods defined in an abstract superclass is restricted to subclass derived from that abstract superclass, which in turn will override these abstract methods

On the other hand, abstract methods declared in an interface can be defined by independent classes not necessarily related to one another through a class hierarchy

The “single-interface, multiple implementations” aspect of runtime polymorphism that you observed earlier in a class hierarchy is given a much wider import and context through interfaces

**JAVA TUTORIAL**

An interface is defined much like a class. This is a simplified general form of an interface:

```
access interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    type final-varname1 = value;  
    type final-varname2 = value;  
    //...  
    return-type method-nameN(parameter-list);  
    type final-varnameN = value;  
}
```

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. name is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods. Each class that includes such an interface must implement all of the methods.

As the general form shows, variables can be declared inside of interface declarations. They are implicitly final and static, meaning

## JAVA TUTORIAL

they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly public.

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods required by the interface.

The general form of a class that includes the implements clause looks like this:

```
class classname [extends superclass] [implements interface  
[,interface...]] {  
  
    // class-body  
  
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

When you implement an interface method, it must be declared as public. A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

When overriding methods defined in interfaces, there are several rules to be followed –

## JAVA TUTORIAL

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementing interfaces, there are several rules –

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.
- An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface. Just as classes can be inherited, interfaces can also be inherited
- One interface can extend one or more interfaces using the keyword **extends**

**JAVA TUTORIAL**

- When you implement an interface that extends another interface, you should provide implementation for all the methods declared within the interface hierarchy

## Enums

The **Enum in Java** is a data type which contains a fixed set of constants. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters. Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*. Here, we can define an enum either inside the class or outside the class.

Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember:

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods

## JAVA TUTORIAL

- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

### Enum and Inheritance:

- All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
- **toString() method** is overridden in **java.lang.Enum class**, which returns enum constant name.
- enum can implement many interfaces.

### values(), ordinal() and valueOf() methods :

- These methods are present inside **java.lang.Enum**.
- **values() method** can be used to return all values present inside enum.
- Order is important in enums. By using **ordinal() method**, each enum constant index can be found, just like array index.
- **valueOf() method** returns the enum constant of the specified string value, if exists.
- **Java User Input:**
- User input can come in many forms - mouse and keyboard interactions, a network request, command-line arguments, files that are updated with data relevant for a program's execution, etc.

## JAVA TUTORIAL

- We're going to focus on keyboard input via something called **the standard input stream**. You may recognize it as Java's System.in.
- We're going to use Scanner class to make our interaction with the underlying stream easier. Since Scanner has some downfalls, we'll also be using the BufferedReader and InputStreamReader classes to process the System.in stream.
- In the end, we'll *Decorate* the InputStream class and implement our own custom UncloseableInputStream to handle issues with the Scanner class.
- The Scanner class is used to get user input, and it is found in the java.util package.
- To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings:
- This Scanner instance can now scan and parse booleans, integers, floats, bytes and Strings.

Java **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package. It is like the Vector in C++.

## JAVA TUTORIAL

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

It inherits the AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Hierarchy of ArrayList class:

Java ArrayList class extends AbstractList class which implements List interface. The List interface extends the Collection and Iterable interfaces in hierarchical order.

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable
```

## JAVA TUTORIAL

# Java Date and Time

Java does not have a built-in Date class, but we can import the `java.time` package to work with the date and time API. The package includes many date and time classes.

Java has introduced a new Date and Time API since Java 8. The `java.time` package contains Java 8 Date and Time classes.

- `java.time.LocalDate` class
- `java.time.LocalTime` class
- `java.time.LocalDateTime` class
- `java.time.MonthDay` class
- `java.time.OffsetTime` class
- `java.time.OffsetDateTime` class
- `java.time.Clock` class
- `java.time.ZonedDateTime` class
- `java.time.ZoneId` class
- `java.time.ZoneOffset` class
- `java.time.Year` class
- `java.time.YearMonth` class
- `java.time.Period` class

## JAVA TUTORIAL

- `java.time.Duration` class
- `java.time.Instant` class
- `java.time.DayOfWeek`\_enum
- `java.time.Month`\_enum

### ***Java LocalDate class***

Java LocalDate class is an immutable class that represents Date with a default format of yyyy-MM-dd. It inherits Object class and implements the ChronoLocalDate interface

Java LocalDate class declaration

Let's see the declaration of `java.time.LocalDate` class.

1. **public final class** `LocalDate` **extends** Object
2. **implements** Temporal, TemporalAdjuster, ChronoLocalDate, Serializable

### ***Java LocalTime class***

Java LocalTime class is an immutable class that represents time with a default format of hour-minute-second. It inherits Object class and implements the Comparable interface.

Java LocalTime class declaration

Let's see the declaration of `java.time.LocalTime` class.

1. **public final class** `LocalTime` **extends** Object

## JAVA TUTORIAL

2. **implements** Temporal, TemporalAdjuster, Comparable<LocalTime>, Serializable

### **Java LocalDateTime class**

Java LocalDateTime class is an immutable date-time object that represents a date-time, with the default format as yyyy-MM-dd-HH-mm-ss.zzz. It inherits object class and implements the ChronoLocalDateTime interface.

Java LocalDateTime class declaration

Let's see the declaration of java.time.LocalDateTime class.

1. **public final class** LocalDateTime **extends** Object
2. **implements** Temporal, TemporalAdjuster, ChronoLocalDateTime<LocalDate>, Serializable

## Methods and Method Overloading

In object-oriented programming, we work with objects. Objects are basic building blocks of a program. Objects consists of data and methods. Methods change the state of the objects created. They are the dynamic part of the objects; the data is the static part.

### **Java method definition**

A method is a code block containing a series of statements. Methods must be declared within a class. It is a good programming practice that methods do only one specific task. Methods bring modularity to programs. Proper use of methods brings the following advantages:

## JAVA TUTORIAL

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

### Java method characteristics

Basic characteristics of methods are:

- Access level
- Return value type
- Method name
- Method parameters
- Parentheses
- Block of statements

Access level of methods is controlled with access modifiers. They set the visibility of methods. They determine who can call the method. Methods may return a value to the caller. If our method returns a value, we declare its data type. If not, we use the void keyword to indicate that our method does not return any value. Method parameters are surrounded by parentheses and separated by commas. Empty parentheses indicate that the method requires no parameters. The method block is surrounded with {} characters. The block contains one or more statements that

## JAVA TUTORIAL

are executed when the method is *invoked*. It is legal to have an empty method block.

### Java method signature

A *method signature* is a unique identification of a method for the Java compiler. The signature consists of a method name, and the type and kind (value, reference, or output) of each of its formal parameters. Method signature does not include the return type.

### Java method names

Any legal character can be used in the name of a method. By convention, method names begin with a lowercase letter. The method names are verbs or verbs followed by adjectives or nouns. Each subsequent word starts with an uppercase character. The following are typical names of methods in Java:

- execute
- findId
- setName
- getName
- checkIfValid
- testValidity

A parameter is a value passed to the method. Methods can take one or more parameters. If methods work with data, we must pass the data to the methods. This is done by specifying them inside the

## JAVA TUTORIAL

parentheses. In the method definition, we must provide a name and type for each parameter.

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, integers and reference data types, such as objects and arrays. When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field.

For example, consider the following Circle class and its setOrigin method:

**JAVA TUTORIAL**

```
public class Circle  
{  
    private int x, y, radius;  
  
    public void setOrigin(int x, int y)  
    {  
        ...  
    }  
}
```

The Circle class has three fields: x, y, and radius. The setOrigin method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names x or y within the body of the method refers to the parameter, *not* to the field. To access the field, you must use a qualified name.

Function power( ) overloaded using different arguments:

```
int power (int x, int y);  
  
float power(float x, float y);
```

Function overloading improves readability if the functions perform closely related tasks. A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. Function selection involves the following steps.

## JAVA TUTORIAL

The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function. If an exact match is not found, the compiler uses the integral promotions to actual arguments, such as, char to int, or float to double to find the match. In case of multiple matches, the compiler will generate an error message.

When an overloaded function is invoked, the compiler chooses the appropriate function by examining the number, data types, and the order of arguments present in that function.

```
int n=power(10,5) //first function is called
```

```
float f=power(10.5,6.3) //second function is called
```

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

**JAVA TUTORIAL**

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.

class OverloadDemo {

    void test() {

        System.out.println("No parameters");

    }

// Overload test for one integer parameter.

    void test(int a) {

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

    }

// Overload test for two integer parameters.

    void test(int a, int b) {

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

    }

// Overload test for a double parameter

    double test(double a) {

        System.out.println("double a: " + a); return a*a;

    }

}
```

**JAVA TUTORIAL**

```
class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob = new OverloadDemo();  
        double result;  
  
        // call all versions of test()  
  
        ob.test();  
  
        ob.test(10);  
  
        ob.test(10, 20);  
  
        result = ob.test(123.25);  
  
        System.out.println("Result of ob.test(123.25): " + result);  
    }  
}
```

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not.

## Scope

Scope refers to the lifetime and accessibility of a variable. How large the scope is depends on where a variable is declared. For

## JAVA TUTORIAL

example, if a variable is declared at the top of a class then it will be accessible to all of the [class methods](#). If it's declared in a method then it can only be used in that method.

### ***Method Scope***

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared

### ***Block Scope***

A block of code refers to all of the code between curly braces {} . Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared

A block of code may exist on its own or it can belong to an if, while or for statement. In the case of for statements, variables declared in the statement itself are also available inside the block's scope.

## ***4 Functions***

---

## JAVA TUTORIAL

### Math Class

The Math class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods. Math defines two double constants: E (approximately 2.72) and PI (approximately 3.14). To perform different numeric calculations in Java, math class has provided several methods. Some of the methods are min(), max(), avg(), sin(), cos() etc.

Below is a list describing several methods that Java math class offers:

**Math.min():** this method returns the **smallest** of two values.

**Math.max():** this method returns the **largest** of two values

**Math.abs():** this method returns the **absolute** value of the value provided.

**Math.round():** this method is used to **round off** the decimal numbers to the nearest value.

**Math.sqrt():** this method returns the **square root** of a number given.

## JAVA TUTORIAL

**Math.cbrt()**: this method returns the **cube root** of a given number.

**Math.floor()**: this method is used to find the **largest integer** value which is less than or equal to the argument and is equal to the double value of a mathematical integer.

**Math.pow()**: this method returns the value of the **first argument** raised to the power second argument provided.

**Math.ceil()**: list method is used to find the **smallest integer** value that is greater than or equal to the argument.

**Math.floorDiv()**: this method is used to find the **largest integer** value which is less than or equal to the quotient.

**Math.random()**: this java random method returns a **double** value that carries a positive sign which is greater than or equal to 0.0 and less than 1.0

**Math.rint()**: this method returns the **double** value that is closest to the given argument.

## JAVA TUTORIAL

**Math.ulp():** this method returns the size of the **ULP** of the argument.

**Math.addExact():** this method is used to return the sum of its arguments and throws an **exception** if the result overflows an integer or a long value.

**Math.subtractExact():** this method returns the difference of the arguments provided and throws an **exception** if the result overflows and an integer value.

**Math.multiplyExact():** this method returns the product of the arguments and throws an **exception** if the result overflows and integer or long value.

**Math.incrementExact():** this method returns the argument which is incremented by one and throws an **exception** if the result overflows and an integer value.

**Math.decrementExact():** this method returns the argument that is decremented by one and throws an **exception** if the result overflows and integer or long value.

**Math.negateExact():** this method returns the negation of the

## JAVA TUTORIAL

argument and throws an **exception** if the result overflows and integer or long value.

### Logarithmic math methods

Here is a list explaining these methods:

**Math.log()**: this method is used to return the **natural logarithm** of double value

**Math.log10()**: this method is used to return the **base 10** logarithm of a double value

**Math.exp()**: this method returns E (**Euler's value**) raised to the power of a double value.

**Math.log1p()**: this method returns the **natural logarithm** of the sum of the argument and also one.

**Math.expm1()**: this method calculates the **power of Euler's number** and subtracts 1 from it.

### Trigonometric math methods

Below mentioned is a list of these methods:

**Math.sin()**: this method returns the **sine value** of a given double value.

## JAVA TUTORIAL

**Math.asin()**: this method returns the **arc sin** value of a given double value.

**Math.cos()**: this method returns the **cos value** of a given double value.

**Math.acos**: this method returns the arc **cosine value** of the given double value.

**Math.tan()**: this method returns the **tangent value** of a given double value.

**Math.atan()**: this method returns the arc **tangent value** of a given double value.

The next segment consists of angular math methods.

### Angular math methods

There are two angular math methods explained below:

**Math.toRadians**: this method is used to convert the specified **degrees** angle to the angle measuring in **radians**.

## JAVA TUTORIAL

**Math.toDegrees:** this method is used to convert the **radians** angle to an equivalent angle measuring in **degrees**.

## String

String, as the name suggest it is the regular sequence of the characters and having done the operations on them. In Java, string is a sequence of characters and is treated as an object of String type. Since strings is available in java as built-in objects it facilitates the comparison of two strings, concatenation, searching etc in a very simple and easy way.

Normally it is very difficult to change the string which is once created so java provides us with the availability of string and string buffer in java.lang through which we can create the copies of the existing string and then can do the changes on the string object that has been created. Both these classes are declared final which means that neither of these classes can be sub classed in any ways.

String constructors:

As any other class, string class also provides support to various number of constructors.

For instance

1. **String s= new String ( );**

Will create an instance of String with no characters.

## JAVA TUTORIAL

### 2. String (char [ ])

Will create a String initialized by an array of characters.

### 3. String(char chars[ ], int startIndex, int numChars)

In this, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use.

### 4. String(String strObj)

In this, strObj is a String object.it will create a String object that contains the same character sequence as another String.

Now let's have a look on the various methods which are related to Strings and how they work with the Strings.

1. String Length It is used to find out the length of any given String, which means the number of characters it holds in it.

length( ) method, **int length( )**

Example:

```
char chars[] = { 'a', 'b', 'c', 'd' };
```

```
String s = new String(chars);
```

## JAVA TUTORIAL

```
System.out.println(s.length());
```

This will give 4 as the output of the statements written above as it contains four characters a,b,c,d.

2 . Special String Operations Java has come up with the special support features to Strings like automatic creation of new String instances from string literals, concatenation of multiple String objects by use of the + operator, and the Conversion of other data types to Strings and Java does all this automatically thus making programming much easier for the Java Programmers.

Let's have a brief understanding of each one of them in the section below:

a. Creation of new String Instances from String Literals: Normally a String instance can be created using the new operator but Java automatically constructs a String Object and thus we can use a String literal to initialize a String Object.

For example,

the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c'};  
String s1 = new String(chars);  
String s2 = "abc"; // use string literal
```

**JAVA TUTORIAL**

This will give two equivalent Strings as the output.

b.String Concatenation Java allows an exception as it gives the permission to implement the + operator on the string for the Concatenation of two strings and producing a String Object as the output of the operation.

For example, the following fragment concatenates three strings:

```
String age = "12";  
String s = "He is " + age + " years" + "old.";  
System.out.println(s);
```

The output of these statements will add up these four Strings and will show:

He is 12 Years old.

c. String Concatenation with Other Data Types. We can also concatenate strings with other types of data. For example, consider this

```
int age = 15;  
String s = "He is " + age + " years old.";  
System.out.println(s);
```

## JAVA TUTORIAL

In this Example although, age is an int rather than another String, but the output produced is the same as before. This is because the int value in age is automatically converted into its string representation within a String object. This string is then concatenated as we saw in the above example.

Hence, the output will be:

He is 15 years old.

3. Character Extraction The Characters can be easily extracted from a String Object.

Each one of them is mentioned below: a. **charAt()** We can directly refer to an individual character and can extract a single character from a String.

String.charAt( ) method.

char charAt(int where)

where is the index of the character that we want to obtain. The value of where must be nonnegative and specify a location within the string.

charAt( ) returns the character at the specified location.

For example

**JAVA TUTORIAL**

```
char ch;  
ch = "abc".charAt(1);  
  
assigns the value “b” to ch.
```

b. **getChars( )** It is used to get more than one character at a time, getChars( ) method.

```
getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

sourceStart specifies the index of the beginning of the substring, sourceEnd specifies an index that is one past the end of the desired substring.

```
public class GetCharsExample {  
  
    public static void main(String[] args) {  
  
        String str = "GetCharsExample";  
        char chars[] = new char[5];  
        str.getChars(1, 6, chars, 0);  
        System.out.println(chars);  
    }  
}
```

## JAVA TUTORIAL

}

### Output

etCha

In java all index starts from 0. So in the above program it will start extracting character from 1st index(second character) and not from 0th index. The end index is 6 so it extracts from second character to seventh character.

c. **getBytes( )** This method stores the characters in an array of bytes and it uses the default character-to-byte conversions provided by the platform.

byte[ ] getBytes( )

It is most useful when we are exporting a String value into an environment that does not support 16-bit Unicode characters.

d. **toCharArray( )** It is used to convert all the characters in a String object into a character array. It returns an array of characters for the entire string.

char[ ] toCharArray( )

4. String Comparison The String class includes several methods that compare strings or substrings within strings.

## JAVA TUTORIAL

Let's discuss each of these methods in brief in the section below:

- a. **equals( ) and equalsIgnoreCase( )** equals ( ) is used to compare two strings for equality.

boolean equals(Object str)

str is the String object which is being compared with the invoking String object.

It will return value as “true” if the Strings have same characters in same order else will give “ False” as the output. Please keep it in mind that it is case sensitive.

We can use equalsIgnoreCase ( ) to do comparison that ignores case differences.

boolean equalsIgnoreCase(String str)

str is the String object being compared with the invoking String object. It,also returns true if the strings contain the same characters in the same order, else false.

```
public class EqualsAndEqualIgnoreCaseExample {
```

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

## JAVA TUTORIAL

```
String str = "EqualsAndEqualIgnoreCaseExample";  
  
String strWithSmallLetters =  
"equalsandequalignorecaseexample";  
  
// equals() is case sensitive  
  
System.out.println(str.equals(strWithSmallLetters));  
  
// equalsIgnoreCase is case insensitive  
  
System.out.println(str.equalsIgnoreCase(strWithSmallLetters));  
}  
}
```

### Output

false

true

5. **regionMatches( )** This method is used to compares a specific region inside a string with another specific region in another string.

It also has two methods one for case sensitive as:

```
boolean regionMatches(int startIndex, String str2, int  
str2startIndex, int numChars)
```

## JAVA TUTORIAL

Another was non case sensitive:

**boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)**

in both the syntaxes startIndex specifies the index at which the region begins within the invoking String object.

Str2 specifies the String being compared .

The index at which the comparison will start within str2 is specified by str2startIndex.

The length of the substring being compared is passed in numChars. In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

**6. startsWith( ) and endsWith( )** As the term illustrates , startsWith( ) method determines whether a given String begins with a specified string. Conversely, endsWith( ) determines whether the String in question ends with a specified string.

startswith ( ): boolean startsWith(String str)

endswith ( ): boolean endsWith(String str)

str is the String being tested. If the string matches, true is returned else false.

**JAVA TUTORIAL**

**7. equals( ) Versus ==** equals( ) method and the == operator have two different functionality for which the programmers normally get confused.

The equals( ) method compares the characters inside a String object while . The == operator compares two object references to see whether they refer to the same instance.

```
public class StringEqualityExample {  
  
    public static void main(String[] args) {  
  
        String str1 = "abc";  
        String str2 = "abc";  
        String str3 = new String("bcd");  
        String str4 = new String("bcd");  
  
        System.out.println(str1 == str2);  
        System.out.println(str1.equals(str2));  
        System.out.println(str3 == str4);  
        System.out.println(str3.equals(str4));  
    }  
}
```

## JAVA TUTORIAL

}

Output generated from the above code is

true

true

false

true

8. **compareTo( )** This method helps us to find out which string is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order.

A string is greater than another if it comes after the other in dictionary order.

**int compareTo(String str)**

str is the String being compared with the invoking String.

The results have the following meanings :

1. Less than zero The invoking string is less than str.  
2. Greater than zero The invoking string is greater than str.  
3. Zero The two strings are equal.

## JAVA TUTORIAL

We can also do the comparison without having the case sensitivity been applied by

```
compareToIgnoreCase( ),  
int compareToIgnoreCase(String str)
```

This method returns the same results as compareTo( ), except that case differences are ignored.

9. Searching Strings We can search a string for a specified character or substring with the below mentioned tow methods:

■ **indexOf( )** it will Search for the first occurrence of a character or substring.

```
int indexOf(int ch)
```

■ **lastIndexOf( )** it will Search for the last occurrence of a character or substring.

```
int lastIndexOf(int ch)
```

ch is the character being sought.\ in both the above written statements.

## JAVA TUTORIAL

To search for the first or last occurrence of a substring, use int indexOf(String str)

**int lastIndexOf(String str)**

str specifies the substring.

10. Modifying a String as it was mentioned earlier too that whenever we want to modify a string we must create a copy with the help of StringBuffer and then do the modifications.

a. **substring( )** A substring can be extracted using substring( ). It has two forms.

1. String substring(int startIndex)

startIndex specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

2. substring( ) allows you to specify both the beginning and ending index of the substring:

String substring(int startIndex, int endIndex) startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

Give Example.

## JAVA TUTORIAL

b. **concat( )** This method helps to concatenate two strings using concat( ), the syntax is : String concat(String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end. It actually does the same function as +.

For example

```
String s1 = "one";  
String s2 = s1.concat("two");
```

puts the string “onetwo” into s2. It generates the same result as the following sequence:

```
String s1 = "one";  
String s2 = s1 + "two";
```

c. **replace( )** This method replaces all occurrences of one character in the invoking string with another character. It's syntax is :

```
String replace(char original, char replacement)
```

original specifies the character to be replaced by the character specified by replacement. The resulting string is returned.

For example, String s = "India".replace('I', 's'); puts the string “ndsa” into s.

## JAVA TUTORIAL

d. **trim( )** This method is used to get the same copy of the string after deletions of the white space if any exists. The syntax for this is :

```
String trim()
```

For example:

String s = " Java language ".trim(); This puts the string "Java language" into s 11. String Buffer: As we have been discussing about the facility provided by String Buffer to represent the characters in java in a growable and modifiable manner. It makes very convenient to the programmers to modify or play around over the strings with the help of String Buffer.

StringBuffer defines these three constructors: 1. StringBuffer( ) 2. StringBuffer(int size) 3. StringBuffer(String str)

String buffer ( ) reserves room for 16 characters without reallocation. StringBuffer( int Size) accepts an integer argument that explicitly sets the size of the buffer. StringBuffer ( String Str) accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

1. **append( )** This method as name defines is used to concatenate the string representation of any other type of data to the end of the invoking StringBuffer object.

## JAVA TUTORIAL

```
StringBuffer append(String str) StringBuffer append(int num)  
StringBuffer append(Object obj)
```

First `String.valueOf( )` method is called for each parameter to obtain its string representation and then the result is appended to the current `StringBuffer` object. The buffer itself is returned by each version of `append( )`.

**2. `charAt( )` and `setCharAt( )`** This method provides us with the value of a single character from a `StringBuffer`

`char charAt(int where)` where specifies the index of the character being obtained.

We can set the value of a character within a `StringBuffer` using `setCharAt( )`.

```
void setCharAt(int where, char ch)
```

where specifies the index of the character being set, and ch specifies the new value of that character.

**3. `delete( )` and `deleteCharAt( )`** These two methods can be used to delete the characters from a `StringBuffer`, their syntaxes are shown below as :

```
StringBuffer delete(int startIndex, int endIndex)
```

**JAVA TUTORIAL**

The delete ( ) method deletes a sequence of characters from the invoking object. startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove. Thus, the substring deleted runs from startIndex to endIndex–1. The resulting StringBuffer object is returned.

`StringBuffer deleteCharAt(int loc)`

The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

**4. ensureCapacity( )** This method is used to set the size of buffer after it has been constructed. Its syntax is :

`void ensureCapacity(int capacity)`

where, capacity specifies the size of the buffer.

**5. getChars( )** This method is used to copy a substring of a StringBuffer into an array, Its syntax is `void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)` sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. of characters in the specified substring.

## JAVA TUTORIAL

6. **insert( )** The insert( ) method inserts one string into another. It is overloaded to accept values of all the simple types, plus Strings and Objects. Like append( ), it calls String.valueOf( ) to obtain the string representation of the value it is called with. This string is then inserted into the invoking StringBuffer object. Few of the forms in which it can be implemented are :

```
StringBuffer insert(int index, String str) StringBuffer insert(int index, char ch) StringBuffer insert(int index, Object obj)
```

index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

7. **length( ) and capacity( )** length( ) method is used to find the length of a StringBuffer and , its total allocated capacity can be found with capacity( ) method. Their syntax is:

```
int length()
```

```
int capacity()
```

8. **replace( )** This method replaces one set of characters with another set inside a StringBuffer object. Its syntax is:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

The substring being replaced is specified by the indexes startIndex and endIndex. Thus, the substring at startIndex through endIndex –

## JAVA TUTORIAL

1 is replaced. The replacement string is passed in str. And returned as an object.

9. **reverse( )** We can reverse the characters within a StringBuffer object using reverse( ), its syntax is : StringBuffer reverse( )

10. **setLength( )** This is used to set the length of the buffer within a StringBuffer object, its syntax is :

```
void setLength(int len)
```

where, len specifies the length of the buffer. This value must be nonnegative.

11. **substring( )** It returns a portion of a StringBuffer. Its syntax is

String substring(int startIndex) It returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object.

```
String substring(int startIndex, int endIndex)
```

It returns the substring that starts at startIndex and runs through endIndex–1

## Object Cloning

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

## JAVA TUTORIAL

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
```

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.

## JAVA TUTORIAL

### 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.

### Call By Value

Functions can be invoked in two ways: **Call by Value** or **Call by Reference**. These two ways are generally differentiated by the type of values passed to them as parameters.

## JAVA TUTORIAL

The parameters passed to function are called *actual parameters* whereas the parameters received by function are called *formal parameters*.

**Call By Value:** In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.

“Call by value” in Java means that argument’s value is copied and is passed to the parameter list of a method. That is, when we call a method with passing argument values to the parameter list, these argument values are copied into the small portion of memory and a copy of each value is passed to the parameters of the called method. Java uses only call by value while passing reference variables as well. It creates a copy of references and passes them as valuable to the methods. As reference points to same address of object, creating a copy of reference is of no harm. But if new object is assigned to reference it will not be reflected.

When these values are used inside the method either for “read or write operations”, we are actually using the copy of these values, not the original argument values which are unaffected by the operation inside the method.

That is, the values of the parameters can be modified only inside the scope of the method but such modification inside the method doesn’t affect the original passing argument.

## JAVA TUTORIAL

When the method returns, the parameters are gone and any changes to them are lost. This whole mechanism is called **call by value or pass by value**.

In call by value, the modification done to the parameter passed does not reflect in the caller's scope while in the call by reference, the modification done to the parameter passed are persistent and changes are reflected in the caller's scope.

## Command Line Arguments

A command-line argument is an information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to `main()`.

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.

**The method parameter of the `main` method contains the command-line arguments in the same order we passed at execution.** If we want to access how much arguments did we get, we only have to check the *length* of the array.

For example, we can print the number of arguments and their value on the standard output:

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

## JAVA TUTORIAL

```
{  
    System.out.println("Argument count: " + args.length);  
  
    for (int i = 0; i < args.length; i++) {  
        System.out.println("Argument " + i + ": " + args[i]);  
    }  
}
```

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]);  
}  
}
```

We compile by > javac CommandLineExample.java

We run by > java CommandLineExample sonoo

**Output:** Your first argument is: sonoo

## JAVA TUTORIAL

# Collections

Collections are basically a mechanism to manipulate and play around with the object references. Collection is mainly used for storing and retrieving several live objects created during the process in your application. As you know each real-world object can be represented and grouped in Java using classes. Collections give you the mechanism of storing many objects in some data structure so that they can be manipulated and retrieved easily.

The classes like dictionary, vector, Stack and properties all had different methods of being implemented and hence there was a need to have unifying them for all the classes and this led to the origin of Collections Framework.

It was basically designed and developed to meet various purposes like:

- High performance of the framework.
- Framework should be able to permit different type of collections to work in a similar manner.
- Extending a collection should always be easy and simple.

Another important feature available in collection is Algorithm. They operate on collections and are defined as static methods within the collection class hence need not to be implemented again and again.

**SHOIB MOHAMMAD**

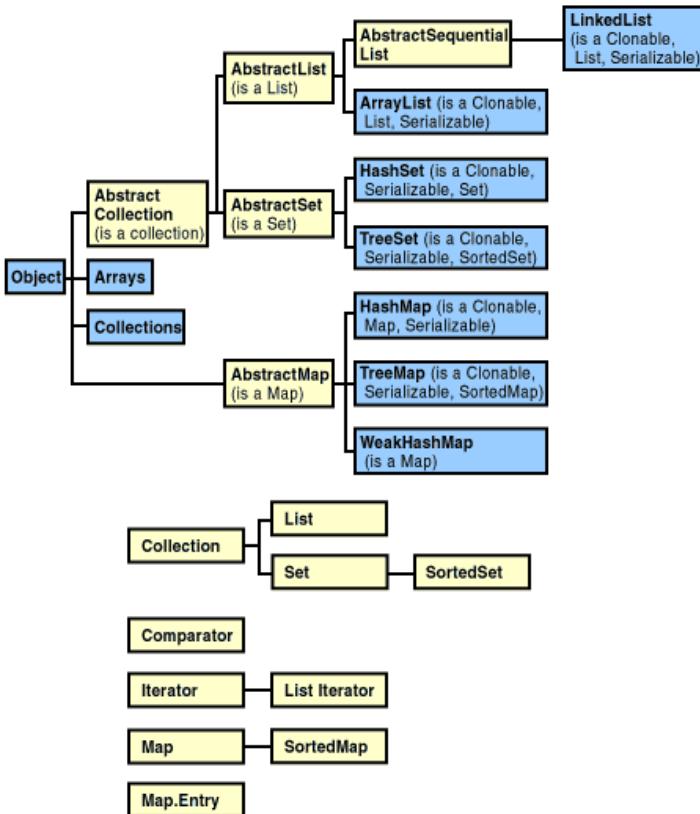
**9840195749**

## **JAVA TUTORIAL**

### ***The Collection Interfaces***

Collection interface provides us a foundation upon which the collection framework is based at and it provides us with concrete classes which can be played around to have various kinds of implementations. It gives us the freedom to play around with the group of objects.

## JAVA TUTORIAL



This provides us different kind of interfaces and it is always at the top of the collections Hierarchy. The collection framework provides with a dozen methods that very efficiently describe the common operations which we are capable of performing the groups of objects.

## JAVA TUTORIAL

We have few very common interfaces like,

Collection it basically helps us to work over the groups of objects and as stated earlier in this section it sits always on the top ranking on collection hierarchy.

**List** Extends Collection to handle sequences (lists of objects)

**Set** Extends Collection to handle sets, which must contain unique elements

**SortedSet** Extends Set to handle sorted sets

**Comparator** as the name suggest defines how to objects are compared.

**Iterator** and **ListIterator** enumerate the objects within a collection.

**Random Access** again as per term indicates that we can have random Access to the elements of the list.

This was just a one liner definition to get you the acquaintance with the interfaces and we will have a look on each one of them in our section below:

We have two broadly categorised interfaces:

## JAVA TUTORIAL

1. Modifiable Interfaces: Collections that allow its methods defined within itself to be played around or in some words to be modified are called modifiable interfaces.
2. Unmodifiable interfaces: Collections which do not provide the flexibility to its methods to be altered or changed belong to the category of unmodifiable interfaces.

A short list of few methods which are defined by collections is :

- a. **boolean add(Object obj)** : This method Adds obj to the invoking collection and returns true if obj was added to the collection else giving if it is already a member of the collection, or if the collection does not allow duplicates.
- b. **boolean addAll(Collection c)** : This method adds all the elements of c to the invoking collection. And gives value as true if the operation succeeded (i.e., the elements were added) else false.
- c. **void clear()** : This method as per its term removes all elements from the invoking collection.
- d. **boolean contains(Object obj)** : This returns true if obj is an element of the invoking collection else returns false.
- e. **boolean containsAll(Collection c)** : Gives Value as true if the invoking collection contains all elements of c. else false.

JAVA TUTORIAL

f. **boolean equals(Object obj)** : Returns true if the invoking collection and obj are equal else returns false.

g. **int hashCode( )** : This method returns the hash code for the invoking collection.

h. **boolean isEmpty( )** : This method returns true if the invoking collection is empty else gives false.

i. **Iterator iterator( )** : It returns an iterator for the invoking collection.

j. **boolean remove(Object obj)** : This method removes one instance of obj from the invoking Collection. It will return value as true if the element is erased else returns false.

k. **boolean removeAll(Collection c)** : This method removes all elements of c from the invoking collection. It Returns true if the collection has been changed or the elements are removed earlier only.

l. **boolean retainAll(Collection c)** : This method removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

m. **int size( )** : This method returns the number of elements held in the invoking collection.

**JAVA TUTORIAL**

# Data Structures

A data structure is a way of storing and organizing data in a computer so that it can be used efficiently. It provides a means to manage large amounts of data efficiently. And efficient data structures are key to designing efficient algorithms.

- Linear
- Hierarchical
- Linear Data Structures
  - Linked List
  - Stacks
  - Queues

A linked list is a linear data structure with the collection of multiple nodes, where each element stores its own data and a pointer to the location of the next element. The last link in a linked list points to null, indicating the end of the chain. An element in a linked list is called a **node**. The first node is called the **head**. The last node is called the **tail**.

**Stack**, an abstract data structure, is a collection of objects that are inserted and removed according to the *last-in-first-out (LIFO)* principle. Objects can be inserted into a stack at any point of time, but only the most recently inserted (that is, “last”) object can be removed at any time.

Listed below are properties of a stack:

## JAVA TUTORIAL

- It is an ordered list in which insertion and deletion can be performed only at one end that is called the *top*
- Recursive data structure with a pointer to its top element
- Follows the ***last-in-first-out (LIFO*** principle
- Supports two most fundamental methods
  - push(e): Insert element e, to the top of the stack
  - pop(): Remove and return the top element on the stack

***Queues*** are also another type of abstract data structure. Unlike a stack, the queue is a collection of objects that are inserted and removed according to the ***first-in-first-out (FIFO)*** principle. That is, elements can be inserted at any point of time, but only the element that has been in the queue the longest can be removed at any time. Listed below are properties of a queue:

- Often referred to as the *first-in-first-out* list
- Supports two most fundamental methods
  - enqueue(e): Insert element e, at the *rear* of the queue
  - dequeue(): Remove and return the element from the *front* of the queue

Queues are used in the asynchronous transfer of data between two processes, CPU scheduling, Disk Scheduling and other situations where resources are shared among multiple users and served on first come first server basis.

- Hierarchical Data Structures

**JAVA TUTORIAL**

- Binary Trees
- Heaps
- Hash Tables

Binary Tree is a hierarchical tree data structures in which *each node has at most two children*, which are referred to as the *left child* and the *right child*. Each binary tree has the following groups of nodes:

- Root Node: It is the topmost node and often referred to as the main node because all other nodes can be reached from the root
- Left Sub-Tree, which is also a binary tree
- Right Sub-Tree, which is also a binary tree

Listed below are the properties of a binary tree:

- A binary tree can be traversed in two ways:
  - *Depth First Traversal*: In-order (Left-Root-Right), Preorder (Root-Left-Right) and Postorder (Left-Right-Root)
  - *Breadth First Traversal*: Level Order Traversal
- Time Complexity of Tree Traversal: O(n)
- The maximum number of nodes at level 'l' =  $2^{l-1}$ .

Applications of binary trees include:

- Used in many search applications where data is constantly entering/leaving

## JAVA TUTORIAL

- As a workflow for compositing digital images for visual effects
- Used in almost every high-bandwidth router for storing router-tables
- Also used in wireless networking and memory allocation
- Used in compression algorithms and many more

Binary Heap is a complete binary tree, which answers to the heap property. In simple terms it is a variation of a binary tree with the following properties:

- *Heap is a complete binary tree:* A tree is said to be complete if all its levels, except possibly the deepest, are complete. This property of Binary Heap makes it suitable to be stored in an array.
- *Follows heap property:* A Binary Heap is either a *Min-Heap* or a *Max-Heap*.
  - Min Binary Heap: For every node in a heap, node's value is *lesser than or equal to* values of the children
  - Max Binary Heap: For every node in a heap, the node's value is *greater than or equal to* values of the children

Popular applications of binary heap include implementing efficient priority-queues, efficiently finding the k smallest (or largest) elements in an array and many more.

## Hash Tables

## JAVA TUTORIAL

Imagine that you have an object and you want to assign a key to it to make searching very easy. To store that key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store data values. However, in cases where the keys are too large and cannot be used directly as an index, a technique called hashing is used.

In hashing, the large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called a **hash table**. A hash table is a data structure that implements a dictionary ADT, a structure that can map unique keys to values.

In general, a hash table has two major components:

1. **Bucket Array:** A bucket array for a hash table is an array A of size N, where each cell of A is thought of as a “bucket”, that is, a collection of key-value pairs. The integer N defines the capacity of the array.
2. **Hash Function:** It is any function that maps each key k in our map to an integer in the range  $[0, N - 1]$ , where N is the capacity of the bucket array for this table.

When we put objects into a hashtable, it is possible that different objects might have the same hashcode. This is called a **collision**. To deal with collision, there are techniques like chaining and open addressing.

## JAVA TUTORIAL

So, these are some basic and most frequently used data structures in Java.

The list Interface extends Collection and it stores a sequence of elements as in list, all these elements can be easily accessed with their serial numbers or sequences and any element can be inserted to the list. But we have the situations when we get UnsupportedOperationException if the collection can't be modified. List interface is implemented by two concrete classes

1. ArrayList
2. LinkedList

### *ArrayLists*

This class extends AbstractList and implements the List interface. This class can be said to be the improved version of the arrays as in arrays we have the limitation of the fixed size which has to be known and defined well in advance and the nothing much can be done with respect to the increase and decrease in the size of the array , so we have got the extended class from collections framework which gives us the flexibility of having the variable array structure which can be dynamically increased or reduced as per the programmers need. An ArrayList is a variable-length array of object references.

ArrayList has three constructors shown here:

- ArrayList()

## JAVA TUTORIAL

- ArrayList(Collection c)
- ArrayList(int capacity)

ArrayList(): This constructor builds an empty array list.

ArrayList(Collection c): This constructor builds an array list that is initialized with the elements of the collection c.

ArrayList (int capacity) : This constructor builds an array list that has the specified initial capacity. The capacity grows automatically as elements are added to an array list.

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class CreateArrayList {
```

```
    public static void main(String args[]) {  
        List list = new ArrayList();  
        list.add("1");  
        list.add("2");  
        list.add("3");  
    }  
}
```

## JAVA TUTORIAL

```
    }  
}
```

The above code will create an ArrayList object and store three String objects in the ArrayList object.

```
import java.util.ArrayList;  
import java.util.List;
```

```
public class DisplayArrayList {  
  
    public static void main(String args[]) {  
        List list = new ArrayList();  
        list.add("1");  
        list.add("2");  
        list.add("3");  
  
        for(int i = 0; i < 3; i++) {  
            System.out.println(list.get(i));  
        }  
    }  
}
```

**JAVA TUTORIAL**

```
 }  
 }
```

Output of the above code

```
1  
2  
3
```

Below code is an example where the arraylist contains many Employee objects. Try this code a see how it works.

```
import java.util.ArrayList;  
import java.util.List;  
  
public class Employees {  
  
    public static void main(String args[]) {  
        List employeesList = new ArrayList();  
        employeesList.add(new Employee("Sam",  
100000));  
        employeesList.add(new Employee("Rohan",  
200000));  
    }  
}
```

**JAVA TUTORIAL**

```
employeesList.add(new Employee("John",  
300000));
```

```
employeesList.add(new Employee("Willey",  
400000));
```

```
int size = employeesList.size();  
for (int i = 0; i < size; i++) {  
    Employee employee = (Employee)  
employeesList.get(i);
```

```
System.out.println(employee.toString());  
}  
}  
}
```

```
class Employee {
```

```
private String name;
```

```
private long sal;
```

JAVA TUTORIAL

```
public Employee(String name, long sal) {
```

```
    this.name = name;
```

```
    this.sal = sal;
```

```
}
```

```
public String getName() {
```

```
    return name;
```

```
}
```

```
public long getSal() {
```

```
    return sal;
```

```
}
```

```
public String toString() {
```

```
    return "Name : " + getName() + "\n" +
```

```
"Salary of " + getSal() + ":" "
```

```
    + getSal();
```

```
}
```

```
}
```

## JAVA TUTORIAL

### ***Linked Lists***

LinkedList Class: This Class extends AbsractSequentialList and implements the List Interface. It as it defines automatically by it's term provides a Linked-list structure.

It has two constructors which are mentioned below:

LinkedList ( ) LinkedList(Collection c)

LinkedList( ): This constructor builds an empty Linked List.

LinkedList(Collection c): This builds a linked list which is initialized with elements of collection .

Try the below code

```
import java.util.LinkedList;
```

```
import java.util.List;
```

```
public class LinkedListExample {
```

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

```
        List linkedList = new LinkedList();
```

**JAVA TUTORIAL**

```
    linkedList.add("1");
    linkedList.add("2");
    linkedList.add("3");
}

}
```

***ArrayList Vs. LinkedList***

There are some differences between an `java.util.ArrayList` and `java.util.LinkedList`. These differences are not so important for any small applications that processes small amount of data. Then they can very important for any small application that processes huge amount of data. These do affect the performance of a application if not analyzed properly.

An empty `LinkedList` takes 2-3 times less memory than an empty `ArrayList`. This becomes important if you plan to keep an adjacency list of nodes for each node in the graph (graph can be like a `TreeSet`. Tree is a type of graph) and you know beforehand that the nodes will have at most one or two adjacent nodes and the total number of nodes in the graph can be quite large.

Although both `ArrayList` and `LinkedList` allow insertion/deletion in the middle through similar operations (ie; by invoking `add(int index)` or `remove(index)`), these operations do not offer you the

## JAVA TUTORIAL

advantage of O(1) insertion/deletion for a LinkedList. For that, you must work through a ListIterator.

The important thing to remember when comparing LinkedList and ArrayList is that linked lists are faster when inserting and removing at random locations in the list multiple times. If you want to add to the end of the list then an ArrayList would be the best choose.

It as per its name, defines a set. It again as List extends collection and it doesn't allow the duplicity of elements. It doesn't define any method of its own like the List Interface.

```
import java.util.LinkedHashSet;
```

```
import java.util.Set;
```

```
public class SetExample {
```

```
    public static void main(String args[]) {  
        Set set = new LinkedHashSet();  
        set.add("1");  
        set.add("1");
```

## JAVA TUTORIAL

```
set.add("2");  
  
System.out.println(set);  
}  
}
```

Output generated after executing the above code

[1, 2]

And it is not like

[1, 1, 2]

### ***How to retrieve elements from a Set***

We have to use Iterators to access the elements in a Set. Though Iterators will be explained in a different section but we will still give you one example which will show how to access the elements in a Set.

```
import java.util.Iterator;  
  
import java.util.LinkedHashSet;  
  
import java.util.Set;
```

**JAVA TUTORIAL**

```
public class SetIteratorExample {
```

```
    public static void main(String args[]) {  
        Set set = new LinkedHashSet();  
        set.add("1");  
        set.add("2");  
        set.add("3");  
        set.add("4");
```

```
        Iterator iter = set.iterator();
```

```
        for(;iter.hasNext();) {  
            String element = (String)  
            iter.next();  
            System.out.println(element);  
        }  
    }
```

## JAVA TUTORIAL

Iterators are mainly inner classes with the concrete class like ArrayList, LinkedList, TreeSet, LinkedHashSet etc. Main purpose of it making as an inner class within these class and not as a different implement class because these implements are specific to their concrete classes.

### *SortedSet Interface*

A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the natural ordering of its elements (see Comparable), or by a Comparator provided at sorted set creation time. Several additional operations are provided to take advantage of the ordering. We can call first( ) method to get the first element and last ( ) to get the last element of the set. headSet ( ) is used to find the subset that starts with the first element in the set and tailSet( ) to get the subset that ends the set.

```
import java.util.Iterator;  
import java.util.SortedSet;  
import java.util.TreeSet;
```

```
public class SortedSetExample {  
  
    public static void main(String[] args) {  
        SortedSet set = new TreeSet(); // TreeSet  
        implements SortedSet.  
    }  
}
```

**JAVA TUTORIAL**

// Otherwise this line  
would have

// given a compiler error  
saying

// could not type cast  
TreeSet to

// SortedSet.

```
set.add("b");
```

```
set.add("c");
```

```
set.add("a");
```

```
for (Iterator iter = set.iterator();  
iter.hasNext(); {  
    String element = (String)  
    iter.next();  
    System.out.println(element);  
}  
}
```

Output of the above code

a

## JAVA TUTORIAL

b

c

### ***TreeSet***

This Class Implements Set Interface that uses Tree structure for the storage of the elements in the set. The objects in this are stored in the ascending order and the access to the elements in this type of arrangement is very quick and efficient and it gives us the feasibility to store a large number of data with this tree structure.

TreeSet has the following constructors:

TreeSet( ): This constructor create the default empty tree.

TreeSet(Collection c) : This builds a tree set that contains the elements of c.

TreeSet(Comparator comp): This constructor constructs an empty tree set that will be sorted according to the comparator specified by comp.

TreeSet(SortedSet ss): This constructor builds a tree set that contains the elements of ss.

```
import java.util.Comparator;
```

```
import java.util.Set;
```

JAVA TUTORIAL

```
import java.util.TreeSet;
```

```
public class TreeSetExample {
```

```
    public static void main(String[] args) {  
        Set set = new TreeSet(new  
        DescendingComparator());
```

```
        set.add("1");
```

```
        set.add("2");
```

```
        set.add("3");
```

```
        set.add("4");
```

```
        set.add("3");
```

```
        System.out.println(set);
```

```
}
```

```
}
```

```
class DescendingComparator implements Comparator {
```

```
    public int compare(Object arg0, Object arg1) {
```

**JAVA TUTORIAL**

```
String str1 = (String) arg0;  
String str2 = (String) arg1;  
return -(str1.compareTo(str2));  
}  
}
```

***HashSet***

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element

```
import java.util.HashSet;  
import java.util.Iterator;  
import java.util.Set;  
  
public class HashSetExample {
```

**JAVA TUTORIAL**

```
public static void main(String args[]) {  
    Set set = new HashSet();  
    set.add("c");  
    set.add("a");  
    set.add(null);  
    set.add("b");  
  
    for(Iterator iter = set.iterator();  
        iter.hasNext();)  
    {  
        String element = (String)  
        iter.next();  
        System.out.println(element);  
    }  
}
```

Output of the above program

null

b

c

**JAVA TUTORIAL**

a

***Linked HashSet***

Hash table and linked list implementation of the Set interface, with predictable iteration order. This implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set.

```
import java.util.Iterator;  
import java.util.LinkedHashSet;  
import java.util.Set;
```

```
public class LinkedHashSetExample {
```

```
    public static void main(String[] args) {  
        Set set = new LinkedHashSet();  
        set.add("c");  
        set.add("b");  
        set.add("a");
```

## JAVA TUTORIAL

```
for(Iterator iter = set.iterator();  
iter.hasNext(); {  
    String element = (String)  
    iter.next();  
    System.out.println();  
}  
}  
}
```

Output of the above program

c  
b  
a

Comparator actually gives a precise understanding of the term “sorted Order” It basically stores the element in exactly the order which is expected to be in sense the natural sorting order. This method defines two other methods compare( ) and equals ( ). Comparators can be passed to a sort method (such as Collections.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as TreeSet or TreeMap).

**JAVA TUTORIAL**

The ordering imposed by a Comparator c on a set of elements S is said to be consistent with equals if and only if `(compare((Object)e1, (Object)e2)==0)` has the same boolean value as `e1.equals((Object)e2)` for every e1 and e2 in S.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

public class ComparatorExample {

    public static void main(String[] args) {
        List employee = new ArrayList();
        employee.add(new
        ComparatorExample().new Employee(2, "Sam", 10000));
        employee.add(new
        ComparatorExample().new Employee(4, "John", 20000));
        employee.add(new
        ComparatorExample().new Employee(1, "Rohan", 30000));
    }
}
```

**JAVA TUTORIAL**

```
employee.add(new  
ComparatorExample().new Employee(3, "Willey", 40000));
```

```
Collections.sort(employee, new  
ComparatorExample().new Employee());
```

```
for (Iterator iter = employee.iterator();  
iter.hasNext();) {  
    Employee emp = (Employee)  
iter.next();
```

```
System.out.println(emp.toString());  
}  
}
```

```
class Employee implements Comparator {
```

```
private int id;
```

```
private String name;
```

JAVA TUTORIAL

```
private long sal;
```

```
public Employee() {
```

```
}
```

```
public Employee(int id, String name, long  
sal) {
```

```
this.id = id;
```

```
this.name = name;
```

```
this.sal = sal;
```

```
}
```

```
public int compare(Object obj1, Object  
obj2) {
```

```
Employee employee1 =
```

```
(Employee) obj1;
```

```
Employee employee2 =
```

```
(Employee) obj2;
```

JAVA TUTORIAL

```
if (employee1.id < employee2.id)
{
    return -1;
}

if (employee1.id > employee2.id)
{
    return 1;
}

return 0;
}

public boolean equals(Object o) {
    if (o instanceof Employee) {
        return false;
    }
}

Employee employee =
(Employee) o;
```

## JAVA TUTORIAL

```
if (employee.id == this.id) {  
    return true;  
}  
  
return true;  
}  
  
public String toString() {  
    return "id: " + this.id + "\n" +  
        "Name: " + this.name + "\n"  
        + "Sal: " +  
    this.sal;  
}  
  
}
```

Output

id: 1

Name: Rohan

## JAVA TUTORIAL

Sal: 30000

id: 2

Name: Sam

Sal: 10000

id: 3

Name: Willey

Sal: 40000

id: 4

Name: John

Sal: 20000

Iterator is of great help when we want any operation to be executed in Cycles, so Iterator will be the best choice if we want the elements of a collection to cycle and display each element. ListIterator extends Iterator.

The following steps are recommended to be followed if we want to implement Iterator:

1. Call the collection's Iterator ( ) method to get to the start of the Iterator.
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
3. Within the loop, obtain each element by calling next( ).

**JAVA TUTORIAL**

We can also implement ListIterator for the collections that element that implement List interface and it works more or less like Iterator.

```
import java.util.Iterator;  
import java.util.LinkedHashSet;  
import java.util.Set;
```

```
public class IteratorExample {
```

```
    public static void main(String args[]) {  
        Set set = new LinkedHashSet();  
        set.add("1");  
        set.add("2");  
        set.add("3");  
        set.add("4");
```

```
        Iterator iter = set.iterator();
```

```
        for(;iter.hasNext();) {
```

**JAVA TUTORIAL**

```
String element = (String)
iter.next();
System.out.println(element);
}
}
```

Maps can be defined as an object that stores associations between keys and values, or key/value pairs. We can find its value if we have the key for it. These keys are unique but their values can be duplicate.

### ***Maps Interfaces***

Let us discuss the interfaces of Maps one by one in our section below:

1. **Map:** This interface maps unique keys to values. This key is nothing but a tool with which we can get this value at a later stage too if required. We can store the value in the map object and we can retrieve this value at any point of time with the help of these unique keys. A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.
2. The Maps basically have two operations: get () and put (). To obtain a value, call get( ), passing the key as an argument and the value is returned as the output to this

**JAVA TUTORIAL**

method and to put a value into a map, use put( ), specifying the key and the value.

3. **SortedMap:** This interface also extends the Map class. It also stores the entries in the ascending order and that's why names as SortedMap. These maps also provide the feasibility to play around with the subMaps using headMap( ), tailMap( ), or subMap( ). To get the first key in the set, call firstKey( ) and call lastKey( ) to get the last key of the submap.
4. **Map.Entry:** This gives us the flexibility to work with a map entry. All the map entries which we get from the entrySet( ) method are basically the elements of Map.Entry Object.

***Map Classes***

There are several classes that provide implementations of the Map Interfaces. The classes that can be used for Maps are explained one by one below in this section.

- a. **AbstractMap:** This class implements most of the Map Interface.
- b. **HashMap:** This class uses a hash table to implement the Map interface thus keeping the execution time constant for the various operations like get( ) and put( ) even for larger group of data. HashMap implements Map and extends AbstractMap. It does not add any methods of its own.

## JAVA TUTORIAL

- c. **TreeMap:** This class implements the Map interface by using a tree. It also assures the storage of the elements in the ascending Order like the HashMap. TreeMap implements SortedMap and extends AbstractMap. It does not define any additional methods of its own.
- d. **LinkedHashMap:** This class extends HashMap Class. LinkedHashMap maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted. You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.
- e. **IdentityHashMap:** This Class implements AbstractMap and is very similar to HashMap except that it makes use of reference equality when it compares the elements. It is not very commonly used.

### HashMap Example

```
import java.util.HashMap;  
import java.util.Map;
```

## JAVA TUTORIAL

```
public class HashMapExample {  
  
    public static void main(String[] args) {  
  
        Map map = new HashMap();  
  
        map.put("1", "John");  
  
        map.put("2", "Ron");  
  
        map.put("3", "Scot");  
  
        map.put("4", "Teri");  
  
        map.put("5", "Sally");  
  
  
        System.out.println("Name : " + map.get("2"));  
        System.out.println("Contains Key : " + map.containsKey("3"));  
        System.out.println("Contains Value : " +  
            map.containsValue("John"));  
  
        System.out.println(map.keySet());  
    }  
}
```

Output

Name : Ron

## JAVA TUTORIAL

Contains Key : true

Contains Value : true

[3, 5, 2, 4, 1]

In a HashMap elements are not sorted in a order. Elements are placed in the HashMap according to the hash value of the Key. It is basically to increase the performance of the HashMap so that elements can be easily and quickly be accessed at any point of time. But java provides one more kind of HashMap which is called LinkedHashMap which maintains the insertion order. When you call keySet() on it, returns the keys according to the insertion order. The example below will help you to understand this important point. We will use the same above code but with LinkedHashMap implementation.

```
import java.util.LinkedHashMap;  
import java.util.Map;  
  
public class LinkedHashMapExample {  
  
    public static void main(String[] args) {  
        Map map = new LinkedHashMap();  
        map.put("1", "John");  
        map.put("2", "Ron");  
    }  
}
```

## JAVA TUTORIAL

```
map.put("3", "Scot");
map.put("4", "Teri");
map.put("5", "Sally");

System.out.println("Name : " + map.get("2"));
System.out.println("Contains Key : " + map.containsKey("3"));
System.out.println("Contains           Value : " +
map.containsValue("John"));

System.out.println(map.keySet());
}
}
```

### Output

Name : Ron

Contains Key : true

Contains Value : true

[1, 2, 3, 4, 5]

### TreeMap Example

```
import java.util.Map;
import java.util.TreeMap;
```

JAVA TUTORIAL

```
public class TreeMapExample {
```

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

```
        Map map = new TreeMap();
```

```
        map.put("2", "John");
```

```
        map.put("1", "Ron");
```

```
        map.put("5", "Scot");
```

```
        map.put("4", "Teri");
```

```
        map.put("3", "Sally");
```

```
        System.out.println("Name : " + map.get("2"));
```

```
        System.out.println("Contains Key : " + map.containsKey("3"));
```

```
        System.out.println("Contains           Value :           "+  
        map.containsValue("John"));
```

```
        System.out.println(map.keySet());
```

```
}
```

```
}
```

## JAVA TUTORIAL

### Output

Name : John

Contains Key : true

Contains Value : true

[1, 2, 3, 4, 5]

There is one Legacy Interface called as Enumeration. This interface defines the methods by which we can enumerate the elements in a collection of objects. It is used by several methods defined by the legacy classes.

The Legacy Classes which are defined by Java.util are discussed below :

1. Dictionary
2. Hashtable
3. Properties
4. Vector
5. Stack

The brief explanation about each of these five classes are taken as under:

1. **Dictionary:** It is an abstract class that denotes a key or a value storage space very much like the Map. Here we can store the value in the dictionary instead of the Map if we have the key and the value of it. And we can anytime retrieve this value

**JAVA TUTORIAL**

with the help pf this unique key. It is basically considered as Obsolete as all its features are available in the improved and advanced form in the form of a Map. We can use put ( ) method to add a key and vaule , a get ( ) method to find the value for any key. The keys and the values can be returned as an enumeration by the keys ( ) method and elements ( ) method. We can also delete the key or valu pair by using the remove ( ) method.

2. **Hashtable:** It is very much similar to hashMap if compared as it also stores the key/value pairs in the table. While Using this we can mention an object that is used as a key, and the value that we want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

A hash table can only store objects that override the hashCode( ) and equals( ) methods that are defined by Object. The hashCode( ) method must compute and return the hash code for the object.

```
import java.util.Hashtable;  
import java.util.Iterator;  
import java.util.Set;
```

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

**JAVA TUTORIAL**

```
Hashtable table = new Hashtable();
```

```
table.put("1", "Pacific Ocean");
```

```
table.put("2", "Atlantic Ocean");
```

```
table.put("3", "India Ocean");
```

```
table.put("4", "Arctic Ocean");
```

```
Set keySet = table.keySet();
```

```
for(Iterator iterator = keySet.iterator(); iterator.hasNext(); {
```

```
    String oceanName = (String) iterator.next();
```

```
    System.out.println(table.get(oceanName));
```

```
}
```

```
}
```

```
}
```

1. **Properties:** It is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. It defines the following instance Variable: Properties defaults; It holds a default property list which is related with a Properties Object.

## JAVA TUTORIAL

Properties defines these two constructors:

- Properties( ) Properties(Properties propDefault)
- Properties ( ): this creates a Properties object that has no default values. Properties(Properties propDefault) : It creates an object that uses propDefault for its default values.

We can specify a default property that will be returned if no value is associated with a certain key.

```
import java.io.FileInputStream;  
import java.io.FileNotFoundException;  
import java.io.IOException;  
import java.util.Properties;
```

```
public class PropertiesExample {  
  
    public static void main(String[] args) throws  
FileNotFoundException, IOException {
```

```
        FileInputStream inputStream = new  
FileInputStream("example.properties");  
  
        Properties properties = new Properties();  
  
        properties.load(inputStream);
```

## JAVA TUTORIAL

```
System.out.println(properties.get("example3"));

System.out.println(properties.get("example1"));

}

}
```

### example.properties

example1=Base Ball

example2=CrickeT

example3=Hockey

example4=Football

example5=Rugby

example6=Lawn Tennis

2. **Vector:** It implements a dynamic array very much similar to the ArrayList. It has few differences as it is synchronized and contains many legacy methods. It basically extends the AbstractList Class and implements the List Interface.

All vectors start with an initial capacity. After this initial capacity is reached, the next time that we attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations

**JAVA TUTORIAL**

that must take place. This reduction is important, because allocations are costly in terms of time. The amount of extra space allocated during each reallocation is determined by the increment that we specify at the time of creating the vector.

If we don't specify an Increment, the vector's size is doubled by each allocation cycle. The increment value is stored in capacityIncrement. The number of elements currently in the vector is stored in elementCount. The array that holds the vector is stored in elementData.

Few Vector constructors are:

- `Vector( )` `Vector(int size)` `Vector(int size, int incr)`  
`Vector(Collection c)`

Each one of them is briefed as under:

- `Vector( )`: It creates a default vector and this vector has the initial size of 10.
- `Vector(int size)`: It creates a vector whose initial capacity is specified by size.
- `Vector(int size, int incr)`: It creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is moved upward.
- `Vector(Collection c)`: It creates a vector that contains the elements of collection c.

**JAVA TUTORIAL**

```
import java.util.Iterator;
import java.util.Vector;

public class VectorExample {

    public static void main(String[] args) {

        Vector vector = new Vector();
        vector.add("Pacific Ocean");
        vector.add("Atlantic Ocean");
        vector.add("India Ocean");
        vector.add("Arctic Ocean");

        for(Iterator iterator = vector.iterator(); iterator.hasNext();) {
            String oceanName = (String) iterator.next();
            System.out.println(oceanName);
        }
    }
}
```

## JAVA TUTORIAL

**Stack:** It is a sub class of Vector which implements a standard last-in, first-out stack. It defines only a default constructor which creates an empty Stack and it includes all the methods defined by vector. We can call pop( ) method to remove and return the top. An EmptyStackException is thrown if you call pop( ) when the invoking stack is empty. We can use peek( ) to return, but not remove, the top object.

The empty( ) method returns true if nothing is on the stack. The search( ) method determines whether an object exists on the stack, and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several Integer objects onto it, and then pops them off again.

## Java Generics

The term generics means parameterized types. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

## JAVA TUTORIAL

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

### Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

- 1) **Type-safety:** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- 2) **Type casting is not required:** There is no need to typecast the object.
- 3) **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

**Following are the rules to define Generic Methods –**

## JAVA TUTORIAL

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

**JAVA TUTORIAL**

## *5 Exception Handling*

---

Exception Handling is a mechanism provided by Java to make programming a very simple, easy and efficient programming language as it provides us with the mechanism through which we can handle the run time errors at the time of coding only.

This Exception handling mechanism has reduced the efforts which the programmer was supposed to handle manually, thus making programming a very cumbersome and tedious task.

### **Java Exceptions**

The exception in Java is basically an object which describes an exceptional condition which is termed as error. Whenever an error occurs in the code, we immediately get an object of that exception created and it is thrown in the method which has actually caused the error in the code. Now, either that method may choose to handle that exception by itself or it may pass it on to other methods in the program.

The basic idea is to catch the exception and handle it properly so that it doesn't cause any kind of trouble to the execution of the rest of the program. These exceptions can be created either due to some manual error or during run-time of the execution of the code, it can arise due to any syntactical error or may be logical error and it becomes very important to the programmer to make sure that all

## JAVA TUTORIAL

the exceptions are caught properly and treated well for the smooth execution of the program.

This is the general form of an exception-handling block:

```
try {  
  
    // block of code to monitor for errors  
  
}  
  
catch (Exception1 exOb) {  
  
    // exception handler for ExceptionType1  
  
}  
  
catch (Exception2 exOb) {  
  
    // exception handler for ExceptionType2  
  
}  
  
finally {  
  
    // block of code to be executed before try block ends  
  
}
```

Here, `ExceptionType` is the type of exception that has occurred.

JAVA TUTORIAL

## Types of Exception

Java manages these Exceptions by five keywords:

1. **Try and Catch**
2. **Throw**
3. **Throws**
4. **Finally**

We will take a look on each one of them in the section below as we proceed so as to have proper understanding of each of them.

Throwable is a built-in super class in Java which stands on the top most level of exception class hierarchy and all other are its sub classes. Below this we have Run time Exceptions which are basically defined automatically and have to be handled immediately as and when they arrive. e.g Division by zero.

Other type of Exception is termed as “Error”, the one which is not expected to arise during normal programming environment and are to be handled very diligently.

Let's take each of the five ways in brief:

### 1. Try and Catch

Although, Java provides us with a default exception handler but it is always beneficial to handle these errors or exceptions manually as it facilitates in two ways, first we get to get the chance to fix the

## JAVA TUTORIAL

error on our own and second that it helps us to control and avoid the automatic termination of the program at any point of time.

It can be done by putting the segment of the code in the try block which has the possibility of having the error and then immediately putting the catch block next to try block with the type of exception to be handled along with a user-friendly message or the treatment you want to give to that exception.

```
import java.io.FileInputStream;  
import java.io.FileNotFoundException;
```

```
public class TryCatchExample {
```

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

```
        try {
```

*// here we need to handle the scenario where file may not exist  
on the system.*

*// So what happens in that case. In such cases your program will  
stop abruptly.*

*// If you want that it should not stop abruptly then you need to  
handle that*

**JAVA TUTORIAL**

// scenarion using try and catch block.

```
FileInputStream inputStream = new  
FileInputStream("this_file_does_not_exist.txt");  
}  
  
catch (FileNotFoundException e) {  
    e.printStackTrace();  
}  
  
}  
}
```

In the practical scenario, we can have multiple catches in one block of code and we can havr multiple catch clauses with respect to all exceptions. At the time of exception handling the catch clause which matches the type of exception will be executed first and rest all the catch clauses will be by passes hence non executed.

## 2. Throw

We get the exceptions not only from the Java run time environment but we do get the exceptions from our programs too with the help of Throw statement.

General syntax of throw is:

## JAVA TUTORIAL

throw new Type of Exception

```
import java.io.FileInputStream;  
import java.io.FileNotFoundException;  
  
public class ThrowExample {  
  
public static void main(String[] args) throws Exception {  
  
    try {  
        FileInputStream inputStream = new  
        FileInputStream("this_file_does_not_exist.txt");  
    }  
  
    catch (FileNotFoundException e) {  
        e.printStackTrace();  
        throw new Exception(e);  
    }  
}  
}
```

## JAVA TUTORIAL

Throw object of superclass throwable or a sub class of it.

Whenever we get the throw statement in the program the execution is stopped immediately after this and the nearest possible try block is checked to find out if there is any catch statement that may match the type of exception which has come, if it doesn't match, the control goes to another try and so on. Finally, if any of the try doesn't match the exception it is handled by the Default Java Exception handler and the program is halted at that place itself.

### 1. **Throws**

It becomes very important at the time of programming that if any exception is created by any method which that method cannot handle it has to be specifically intimated to the programmers so that they are aware of it before they call that method in their block of code and it is done with the help of Throws statement been written in that particular method itself.

All the exceptions which the method can throw without handling them should be displayed properly in the form of a list should be declared in the throws clause else it will lead to a compile time error.

General form of a method declaration that includes a throws clause is

**JAVA TUTORIAL**

type method-name(parameter-list) throws exception-list { // body of method } Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ThrowsExample {

    public static void main(String[] args) throws Exception {
        try {
            FileInputStream inputStream = new
FileInputStream("this_file_does_not_exist.txt");
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
            throw new Exception(e);
        }
    }
}
```

**JAVA TUTORIAL****2. Finally**

Whenever we get any type of exception in our code, we try to handle that exception but, in that process, it happens that we always skip few lines of our code as the normal flow of the program gets disturbed and it may sometimes put us to serious situations. In order to handle this scenario, we have one keyword called Finally which helps us to address this problem.

finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

```
import java.io.FileInputStream;  
import java.io.FileNotFoundException;
```

```
public class FinallyExample {
```

## JAVA TUTORIAL

```
public static void main(String[] args) throws Exception {  
  
    try {  
  
        FileInputStream inputStream = new  
FileInputStream("this_file_does_not_exist.txt");  
  
    }  
  
    catch (FileNotFoundException e) {  
  
        e.printStackTrace();  
  
        throw new Exception(e);  
  
    }  
  
    finally {  
  
        //finally will always be executed.  
  
    }  
  
}  
}
```

### Java Built- in Exceptions

## JAVA TUTORIAL

Java provides various built-in classes to the programmer which facilitates coding in Java. The most common exceptions are subclasses of the standard type `RuntimeException`. They are not required to be manually included in any method's throws list. They are basically defined as unchecked exceptions because they are not checked by the compiler to see if a method handles or throws these exceptions.

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

### 1. **ArithmaticException**

It is thrown when an exceptional condition has occurred in an arithmetic operation.

### 2. **ArrayIndexOutOfBoundsException**

It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

### 3. **ClassNotFoundException**

This Exception is raised when we try to access a class whose definition is not found

### 4. **FileNotFoundException**

This Exception is raised when a file is not accessible or does not open.

### 5. **IOException**

It is thrown when an input-output operation failed or interrupted

## JAVA TUTORIAL

### 6. **InterruptedException**

It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

### 7. **NoSuchFieldException**

It is thrown when a class does not contain the field (or variable) specified

### 8. **NoSuchMethodException**

It is thrown when accessing a method which is not found.

### 9. **NullPointerException**

This exception is raised when referring to the members of a null object. Null represents nothing

### 10. **NumberFormatException**

This exception is raised when a method could not convert a string into a numeric format.

### 11. **RuntimeException**

This represents any exception which occurs during runtime.

### 12. **StringIndexOutOfBoundsException**

It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

## User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’. Following steps are followed for the creation of user-defined Exception.

## JAVA TUTORIAL

- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:

*class MyException extends Exception*

- We can write a default constructor in his own exception class.

*MyException(){}*

- We can also create a parameterized constructor with a string as a parameter.
- We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

*MyException(String str) { super(str); }*

- To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

## JAVA TUTORIAL

```
MyException me = new MyException("Exception details"); throw  
me;
```

### Checked Exceptions

You know that the package `java.lang` is implicitly imported into all Java programs. The `java.lang` package contains the exception classes, most of which are of type `RunTimeException`. Therefore all these exception classes are automatically available to our programs. The compiler checks to see if checked exceptions are specified or not. If they are not specified, then the compiler gives a compilation error. Following are some of the checked exceptions:

- `ClassNotFoundException`
- `NoSuchMethodException`
- `InterruptedException`
- `IllegalAccessException`
- `InstantiationException`.

Why did the designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Any Exception that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and return value.

## JAVA TUTORIAL

Although the Java language has provided a list of exceptions to handle various situations, it has also provided the facility of creating your own exceptions.

There can be situations when you want to handle a specific situation and the available exceptions cannot fulfill your need.

The following are some of the unchecked exceptions:

- `ArithmaticException`
- `ArrayIndexOutOfBoundsException`
- `NegativeArraySizeException`
- `ClassCastException`
- `NullPointerException`
- `SecurityException`

## Regular Exceptions

*Regular expressions* are a way to describe a set of strings based on common characteristics shared by each string in the set. They can be used to search, edit, or manipulate text and data. You must learn a specific syntax to create regular expressions — one that goes beyond the normal syntax of the Java programming language. Regular expressions vary in complexity, but once you understand the basics of how they're constructed, you'll be able to decipher (or create) any regular expression.

## JAVA TUTORIAL

The `java.util.regex` package primarily consists of three classes: `Pattern`, `Matcher`, and `PatternSyntaxException`.

- A `Pattern` object is a compiled representation of a regular expression. The `Pattern` class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a `Pattern` object. These methods accept a regular expression as the first argument; the first few lessons of this trail will teach you the required syntax.
- A `Matcher` object is the engine that interprets the pattern and performs match operations against an input string. Like the `Pattern` class, `Matcher` defines no public constructors. You obtain a `Matcher` object by invoking the `matcher` method on a `Pattern` object.
- A `PatternSyntaxException` object is an unchecked exception that indicates a syntax error in a regular expression pattern.

## Flags

The `pattern` class of `java.regex` package is a compiled representation of a regular expression.

The **compile()** method of this class accepts a string value representing a regular expression and returns a **Pattern** object, the following is the signature of this method.

***static Pattern compile(String regex)***

## JAVA TUTORIAL

Another variant of this method accepts an integer value representing flags, following is the signature of the compile method with two parameters.

***static Pattern compile(String regex, int flags)***

### **Creating a Pattern with Flags**

The Pattern class defines an alternate compile method that accepts a set of flags affecting the way the pattern is matched. The flags parameter is a bit mask that may include any of the following public static fields:

- Pattern.CANON\_EQ Enables canonical equivalence. When this flag is specified, two characters will be considered to match if, and only if, their full canonical decompositions match. The expression "a\u030A", for example, will match the string "\u00E5" when this flag is specified. By default, matching does not take canonical equivalence into account. Specifying this flag may impose a performance penalty.
- Pattern.CASE\_INSENSITIVE Enables case-insensitive matching. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case-insensitive matching can be enabled by specifying the UNICODE\_CASE flag in conjunction with this flag. Case-insensitive matching can also be enabled via the embedded flag expression (?i).

## JAVA TUTORIAL

Specifying this flag may impose a slight performance penalty.

- Pattern.COMMENTS Permits whitespace and comments in the pattern. In this mode, whitespace is ignored, and embedded comments starting with # are ignored until the end of a line. Comments mode can also be enabled via the embedded flag expression (?x).
- Pattern.DOTALL Enables dotall mode. In dotall mode, the expression . matches any character, including a line terminator. By default this expression does not match line terminators. Dotall mode can also be enabled via the embedded flag expression (?s). (The s is a mnemonic for "single-line" mode, which is what this is called in Perl.)
- Pattern.LITERAL Enables literal parsing of the pattern. When this flag is specified then the input string that specifies the pattern is treated as a sequence of literal characters. Metacharacters or escape sequences in the input sequence will be given no special meaning. The flags CASE\_INSENSITIVE and UNICODE\_CASE retain their impact on matching when used in conjunction with this flag. The other flags become superfluous. There is no embedded flag character for enabling literal parsing.
- Pattern.MULTILINE Enables multiline mode. In multiline mode the expressions ^ and \$ match just after or just before, respectively, a line terminator or the end of the input sequence. By default these expressions only

**JAVA TUTORIAL**

match at the beginning and the end of the entire input sequence. Multiline mode can also be enabled via the embedded flag expression (?m).

- Pattern.UNICODE\_CASE Enables Unicode-aware case folding. When this flag is specified then case-insensitive matching, when enabled by the CASE\_INSENSITIVE flag, is done in a manner consistent with the Unicode Standard. By default, case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case folding can also be enabled via the embedded flag expression (?u). Specifying this flag may impose a performance penalty.
- Pattern.UNIX\_LINES Enables UNIX lines mode. In this mode, only the '\n' line terminator is recognized in the behavior of ., ^, and \$. UNIX lines mode can also be enabled via the embedded flag expression (?d).

## Expression Patterns

The Java Pattern class (java.util.regex.Pattern), is the main access point of the Java regular expression API. Whenever you need to work with regular expressions in Java, you start with Java's Pattern class.

Working with regular expressions in Java is also sometimes referred to as *pattern matching in Java*. A regular expression is also sometimes referred to as a *pattern* (hence the name of the

## JAVA TUTORIAL

Java Pattern class). Thus, the term pattern matching in Java means matching a regular expression (pattern) against a text using Java.

The Java Pattern class can be used in two ways. You can use the Pattern.matches() method to quickly check if a text (String) matches a given regular expression. Or you can compile a Pattern instance using Pattern.compile() which can be used multiple times to match the regular expression against multiple texts. Both the Pattern.matches() and Pattern.compile() methods

### **Pattern.matches()**

The easiest way to check if a regular expression pattern matches a text is to use the static Pattern.matches() method. Here is a Pattern.matches() example in Java code:

```
import java.util.regex.Pattern;

public class PatternMatchesExample {

    public static void main(String[] args) {

        String text =
            "This is the text to be searched " +
            "for occurrences of the pattern./";

        String pattern = ".*is.*";

        boolean matches = Pattern.matches(pattern, text);

        System.out.println("matches = " + matches);

    }
}
```

## JAVA TUTORIAL

```
}
```

This Pattern.matches() example searches the string referenced by the text variable for an occurrence of the word "is", allowing zero or more characters to be present before and after the word (the two .\* parts of the pattern).

The Pattern.matches() method is fine if you just need to check a pattern against a text a single time, and the default settings of the Pattern class are appropriate.

If you need to match for multiple occurrences, and even access the various matches, or just need non-default settings, you need to compile a Pattern instance using the Pattern.compile() method.

### **Pattern.compile()**

If you need to match a text against a regular expression pattern more than one time, you need to create a Pattern instance using the Pattern.compile() method. Here is a Java Pattern.compile() example:

```
import java.util.regex.Pattern;

public class PatternCompileExample {

    public static void main(String[] args) {
        String text =
            "This is the text to be searched " +
            "for occurrences of the http:// pattern.";

        String patternString = ".*http://.*";
    }
}
```

## JAVA TUTORIAL

```
Pattern pattern = Pattern.compile(patternString);  
}  
}
```

You can also use the `Pattern.compile()` method to compile a `Pattern` using special flags. Here is a Java `Pattern.compile()` example using special flags:

```
Pattern pattern = Pattern.compile(patternString,  
Pattern.CASE_INSENSITIVE);
```

The Java `Pattern` class contains a list of flags (int constants) that you can use to make the `Pattern` matching behave in certain ways. The flag used above makes the pattern matching ignore the case of the text when matching.

### **Pattern.matcher()**

Once you have obtained a `Pattern` instance, you can use that to obtain a `Matcher` instance. The `Matcher` instance is used to find matches of the pattern in texts. Here is an example of how to create a `Matcher` instance from a `Pattern` instance:

```
Matcher matcher = pattern.matcher(text);
```

The `Matcher` class has a `matches()` method that tests whether the pattern matches the text. Here is a full example of how to use the `Matcher`:

```
import java.util.regex.Pattern;  
import java.util.regex.Matcher;
```

**JAVA TUTORIAL**

```
public class PatternMatcherExample {  
    public static void main(String[] args) {  
        String text =  
            "This is the text to be searched " +  
            "for occurrences of the http:// pattern.";  
  
        String patternString = ".*http://.*";  
  
        Pattern pattern = Pattern.compile(patternString,  
Pattern.CASE_INSENSITIVE);  
  
        Matcher matcher = pattern.matcher(text);  
  
        boolean matches = matcher.matches();  
  
        System.out.println("matches = " + matches);  
    }  
}  
  
Pattern.split()
```

The `split()` method in the `Pattern` class can split a text into an array of `String`'s, using the regular expression (the pattern) as delimiter. Here is a Java `Pattern.split()` example:

```
import java.util.regex.Pattern;  
  
public class PatternSplitExample {  
  
    public static void main(String[] args) {
```

## JAVA TUTORIAL

```
String text = "A sep Text sep With sep Many sep Separators";
```

```
String patternString = "sep";
```

```
Pattern pattern = Pattern.compile(patternString);
```

```
String[] split = pattern.split(text);
```

```
System.out.println("split.length = " + split.length);
```

```
for(String element : split){
```

```
    System.out.println("element = " + element);
```

```
}
```

```
}
```

```
}
```

This `Pattern.split()` example splits the text in the `text` variable into 5 separate strings. Each of these strings are included in the String array returned by the `split()` method. The parts of the text that matched as delimiters are not included in the returned String array.

### **Pattern.pattern()**

**JAVA TUTORIAL**

The pattern() method of the Pattern class simply returns the pattern string (regular expression) that the Pattern instance was compiled from. Here is an example:

```
import java.util.regex.Pattern;  
  
public class PatternPatternExample {  
  
    public static void main(String[] args) {  
  
        String patternString = "sep";  
        Pattern pattern = Pattern.compile(patternString);  
  
        String pattern2 = pattern.pattern();  
    }  
}
```

## Metacharacters

Metacharacters are characters which are having special meanings in Java regular expression.

Following metacharacters are supported in Java Regular expressions.

**JAVA TUTORIAL**

Regular Expression	Description
\d	Any digits, short for [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

There are two ways to use metacharacters as ordinary characters in regular expressions.

1. Precede the metacharacter with a backslash (\).
2. Keep metacharacter within \Q (which starts the quote) and \E (which ends it).

**JAVA TUTORIAL**

## ***6 Threads***

---

Java provides the facility of multi-threaded programming which means that java enables the feature in which two or more parts of a program can run simultaneously without conflicting with each other and each part of the program is called “thread”. This multi-threaded program is a very good example of multi-tasking. It helps us to write programs that make the maximum use of CPU time as we keep the CPU busy with various simultaneously running tasks hence providing minimum idle time for the CPU.

Threads exist in several states. A thread can be running. It can be ready to run as soon as it gets that CPU is available. A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

### ***Thread Priorities***

Java provides the priority in the form of integer value to each of its thread which decides how that particular thread will be treated in comparison to the other threads which are running. This priority doesn't really work if there's only single thread which is running. Ideally it is said that the higher priority threads more CPU time but practically the scenario is not like this as the CPU time depends on

## JAVA TUTORIAL

various other factors too apart from the thread priority. This switching from execution of one thread to another on the basis of its priority value is called “Context Switch”.

The basic rules for context switch are as under:

- A thread can voluntarily relinquish control. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

Since java is designed to work in various environments it is very important that the threads which have same level of priority should be controlled once in a while.

To set a thread's priority, use the `setPriority( )` method, which is a member method of Thread.

General syntax of thread priority is **final void setPriority ( Int level)**

Level is the new priority setting for the particular calling thread and this value may range from the **MIN\_PRIORITY** and **MAX\_PRIORITY**.

## JAVA TUTORIAL

We can always find out the current priority setting by calling the `getPriority()` method of the thread by writing this line of code **final int getPriority()**

**Messaging:** Java provides the feature to the threads to communicate with each other with the help of predefined methods which are available with all the objects. It allows a thread to enter a synchronised method on an object and then wait till the other thread communicates it to come out of that.

**Main Thread:** In the Java programming whenever we start a program the first thread which starts its execution first is called the “main thread” of the program. It is a very important thread as it will lead to have other “child” thread running in the program and most of the times it also becomes the last thread to finish the execution of shutdown operations.

The main thread although is created automatically with the starting of the program it can be controlled like other thread by getting the reference of the thread by calling `currentThread()` method.

### ***Thread Methods***

**JAVA TUTORIAL**

Method	Description
<b>setName()</b>	to give thread a name
<b>getName()</b>	return thread's name
<b>getPriority()</b>	return thread's priority
<b>isAlive()</b>	checks if thread is still running or not
<b>join()</b>	Wait for a thread to end
<b>run()</b>	Entry point for a thread
<b>sleep()</b>	suspend thread for a specified time
<b>start()</b>	start a thread by calling run() method

***Creating a Thread***

Java provides two ways of creating the thread:

- We can implement the **Runnable interface**.
- We can also extend the **Thread class**, itself.

Let's have a look at each one of them-

***Runnable Interface***

We can create a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared as under:

**public void run( )**

**JAVA TUTORIAL**

Inside run( ), the code is defined that constitutes the new thread. After the class is created that implements Runnable, an object is instantiated of type Thread from within that class.

```
public class RunnableExample implements Runnable {  
  
    private String threadName;  
  
    public RunnableExample(String name) {  
        threadName = name;  
    }  
  
    public void run() {  
  
        for(int i = 0; i < 1000; i++) {  
            System.out.println("Thread " + threadName + " : " + i);  
        }  
    }  
  
    public static void main(String args[]) {  
        Thread thread1 = new Thread(new RunnableExample("A"));  
    }  
}
```

**JAVA TUTORIAL**

```
thread1.start();  
Thread thread2 = new Thread(new RunnableExample("B"));  
thread2.start();  
}  
}
```

Try the above code. You will find some interesting results every time you execute the code. You can see that we have used start () method. Main purpose of start method is to start the thread. It asks JVM to allocate space and time for the thread so that the thread can execute.

### ***Extending Thread***

Another way of creating the thread is to create a new class which will extend the thread. And later we will create the instance of it.

The class which is been extended should override the run ( ) method and should also call start ( ) method to start the execution of the newly made thread.

```
public class ThreadExample extends Thread {  
  
private String threadName;
```

**JAVA TUTORIAL**

```
public ThreadExample(String name) {  
    threadName = name;  
}  
  
public void run() {  
  
    for(int i = 0; i < 1000; i++) {  
  
        System.out.println("Thread " + threadName + " : " + i);  
    }  
}  
  
public static void main(String args[]) {  
  
    Thread thread = new ThreadExample("A");  
  
    thread.start();  
}  
}
```

***Creating multiple Threads***

It is also possible to have multiple threads which will run simultaneously. This will be illustrated to you with an example:

JAVA TUTORIAL

```
public class ThreadExample extends Thread {
```

```
    private String threadName;
```

```
    public ThreadExample(String name) {
```

```
        threadName = name;
```

```
}
```

```
    public void run() {
```

```
        for(int i = 0; i < 1000; i++) {
```

```
            System.out.println("Thread " + threadName + " : " + i);
```

```
}
```

```
}
```

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

```
        Thread thread1 = new ThreadExample("A");
```

```
        thread1.start();
```

```
        Thread thread2 = new ThreadExample("B");
```

**JAVA TUTORIAL**

```
    thread2.start();  
}  
}
```

**Using isAlive( ) and join( )**

As we have discussed earlier in the heading Main thread that normally it is recommended that the main thread which starts firstly when the program is started should ideally end also in the last thus making sure that all the other child threads have finished their execution before the main thread has ended its execution and this is made possible normally with the sleep( ) function within main( ) but this is not the ideal solution .

Java provides us with a better solution in threads by which we can find out whether all other child threads have finished executing before the main thread and it is done in two ways as mentioned:

1. We can call the isAlive( ) method which will return the value as “true” if the thread is running on which it is called else will give “false” as it’s returning value/
2. Another method is to implement join ( ) function. this method will wait until the thread on which it is called is terminated.

Both these methods are explained in detail with the examples below.

```
public class IsAliveExample {
```

**JAVA TUTORIAL**

```
public static void main(String[] args) {  
    System.out.println(Thread.currentThread().isAlive());  
}  
}
```

The above program will always return true because it is being called within the main thread and which will always be alive during the call of isAlive method.

```
public class IsAliveExample {  
  
    public static void main(String[] args) {  
        Thread th = new MyOwnThread("A");  
        th.start();  
  
        // wait till the thread is alive  
        while(th.isAlive());  
  
        System.out.println("Thread is not alive any more.");  
    }  
}
```

JAVA TUTORIAL

}

```
class MyOwnThread extends Thread {
```

```
    private String threadName;
```

```
    public MyOwnThread(String name) {
```

```
        threadName = name;
```

```
}
```

```
    public void run() {
```

```
        for(int i = 0; i < 1000; i++) {
```

```
            System.out.println("Thread " + threadName + " : " + i);
```

```
        }
```

```
}
```

```
}
```

**JAVA TUTORIAL**

Try the above code. Remember we are checking isAlive() for the thread called th and not the main method method.

```
public class JoinExample {  
  
    public static void main(String[] args) {  
  
        Thread th1 = new MyOwnThread("A");  
        Thread th2 = new MyOwnThread("B");  
  
        try {  
            th1.start();  
            th1.join();  
            th2.start();  
        }  
        catch (InterruptedException e) {  
  
        }  
    }  
}
```

**JAVA TUTORIAL**

```
class MyOwnThread extends Thread {  
  
    private String threadName;  
  
    public MyOwnThread(String name) {  
        threadName = name;  
    }  
  
    public void run() {  
  
        for (int i = 0; i < 1000; i++) {  
            System.out.println("Thread " + threadName + " : " + i);  
        }  
    }  
}
```

The above code guarantees that th2 will be executed only once th1 complete its execution. Remember that if you begin two threads simultaneously the output will never will the same in every execution. But in the above code you can control the output using

## JAVA TUTORIAL

join(). Try executing the same code without using join() you will find your answers.

## Synchronization

It is basically the mechanism which helps two or more threads to share all the available resources in a sequential manner thus making sure that one resource is been used by only one single thread at a time thus avoiding the problem of collision or conflict between the threads and it is done in Java with the concept of “monitor” also named as “semaphore”.

The basic working mechanism of monitor works in a way that it works like a lock and this monitor can be owned by only one thread at a time and until the time this monitor or lock is been released by that particular thread the other threads are said to be in the waiting stage to acquire for that monitor or lock.

In Java we can enter into any monitor or lock by calling the method that has been modified with the synchronized keyword.

General form of the synchronized statement is as

```
synchronized(object) {
```

```
// statements to be synchronized
```

## JAVA TUTORIAL

}

where object is a reference to the object being synchronized.

A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

### Interthread communication

Java implements a very efficient interprocess communication which reduces the CPU's idle time to a very great extent. It is been implemented through wait(), notify() and notifyAll() methods. Since these methods are implemented as final methods, they are present in all the classes.

The basic functionality of each one of them is as under:

- **wait()** acts as a intimation to the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()**.
- **notify()** is used as intimator to wake up the first thread that called **wait()** on the same object.
- **notifyAll()** as the term states wakes up all the threads that called **wait()** on the same object. The highest priority thread will run first.

```
public class WaitNotifyAllExample {
```

JAVA TUTORIAL

```
public static void main(String[] args) {  
  
    try {  
  
        Object o = new Object();  
  
        Thread thread1 = new Thread(new MyOwnRunnable("A", o));  
  
        Thread thread2 = new Thread(new MyOwnRunnable("B", o));  
  
        Thread thread3 = new Thread(new MyOwnRunnable("C", o));  
  
        // synchronized keyword acquires lock on the object.  
  
        synchronized (o) {  
  
            thread1.start();  
  
            // wait till the first thread completes execution.  
  
            // thread should acquire the lock on the object  
  
            // before calling wait method on it. Otherwise it will  
  
            // throw java.lang.IllegalMonitorStateException  
  
            o.wait();  
  
            thread2.start();  
  
            // wait till the second thread completes execution  
  
            o.wait();  
    }  
}
```

JAVA TUTORIAL

```
        thread3.start();  
    }  
  
}  
  
catch (InterruptedException e) {  
    e.printStackTrace();  
}  
  
}  
  
}  
  
class MyOwnRunnable implements Runnable {  
  
    private String threadName;  
  
    private Object o;  
  
    public MyOwnRunnable(String name, Object o) {  
        threadName = name;
```

JAVA TUTORIAL

```
this.o = o;  
}  
  
public void run() {  
  
synchronized (o) {  
    for (int i = 0; i < 1000; i++) {  
        System.out.println("Thread " + threadName + " Count : " + i);  
    }  
    // notify all threads waiting for the object o.  
    // thread should acquire the lock on the object  
    // before calling notify or notifyAll method on it.  
    // Otherwise it will throw java.lang.IllegalMonitorStateException  
    o.notifyAll();  
}  
}
```

**JAVA TUTORIAL**

## Multithreading

### Need for Multithreading

Threads are separate tasks running within a program. If your program consists of a single thread, it can handle only one activity at a time.

Even before you venture into multithreading, it would be appropriate to first of all learn as to, what is multitasking, for, there are lot of conceptual similarities between multitasking and multithreading.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's control. However, multithreaded multitasking is.

A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.

## JAVA TUTORIAL

In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler. In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

## What is Multitasking

Multitasking seemingly creates an illusion of running two or more programs simultaneously, but in reality, it executes only one program at any given point of time. The operating system does this so efficiently that it seems that it is handling more than one program simultaneously. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Multitasking is synonymous with process-based multitasking, whereas multithreading is synonymous with thread-based multitasking. All modern operating systems support multitasking. A process is an executing instance of a program

Process-based multitasking is the feature by which the operating system runs two or more programs concurrently

Multithreaded programs extend the idea of multitasking by taking it one level lower: Individual programs will appear to do multiple tasks at the same time. Each task is usually called a thread, which

## JAVA TUTORIAL

is short for thread of control. Programs that can run more than one thread at once are said to be multithreaded.

In a thread based multitasking environment, the thread is the smallest unit of code that can be dispatched by the thread scheduler. This means that a single program can perform two or more tasks concurrently.

It must be noted though, that, if there are multiple threads in a program, only one thread will be executing at any given point of time given a single-processor architecture.

In multithreading, the thread is the smallest unit of code that can be dispatched by the thread scheduler. A single program can perform two tasks using two threads. Only one thread will be executing at any given point of time given a single-processor architecture. Effectively used, they can prevent the user interface from being tied up while the program is performing a lengthy operation. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time of the CPU can be kept down to a minimum. This is especially important for the interactive, networked Internet environment in which Java operates, because idle time is common. For example, there may be an instruction in a program that is reading a file from a different host on the network.

Knowing that reading from a local file system itself is slow compared to the speed of the CPU, reading from a file on a different host on the network is an even slower process. In a traditional single-threaded environment, the rest of the program waits till this I/O instruction returns. Once the CPU dispatches this

## JAVA TUTORIAL

I/O instruction, it is free. Ideally, this idle time of the CPU should be put to good use by executing some other discrete part of the program.

In the traditional single-threaded environment with reference to the aforesaid example, the CPU remains idle till the I/O instruction returns wasting precious CPU cycles.

## Concurrency

Concurrency is the ability to run several programs or several parts of a program in parallel. If a time-consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program.

A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or

## JAVA TUTORIAL

updating the display. Software that can do such things is known as *concurrent* software.

A *process* runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g., memory and CPU time, are allocated to it via the operating system.

A *thread* is a so-called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data, it stores this data in its own memory cache.

A thread can re-read the shared data.

A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior. Concurrency promises to perform certain task faster as these tasks can be divided into subtasks and these subtasks can be executed in parallel. Of course, the runtime is limited by parts of the task which can be performed in parallel.

Threads have their own call stack, but can also access shared data. Therefore, you have two basic problems, visibility and access problems.

## JAVA TUTORIAL

A visibility problem occurs if thread A reads shared data which is later changed by thread B and thread A is unaware of this change.

An access problem can occur if several threads access and change the same shared data at the same time.

Visibility and access problem can lead to:

- Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks.
- Safety failure: The program creates incorrect data.

A Java program runs in its own process and by default in one thread. Java supports threads as part of the Java language via the Thread code. The Java application can create new threads via this class.

Java 1.5 also provides improved support for concurrency with the `java.util.concurrent` package.

### Deadlock

Deadlock as the name illustrates it is a special type of error that each programmer should make sure to avoid in a multitasking environment. This is a very difficult error to be debugged because it occurs once at a time when two threads time-slice in just the right way and other reason for a deadlock to occur is when it involves more than two words and two synchronized objects at a time.

## JAVA TUTORIAL

It is illustrated in a better way with the help of an example.

Suspending, Resuming and stopping threads: Suspending a thread in java is quite an easy task and similarly we can resume that particular thread also in a very simple way.

The thread can be suspended using **suspend ()** method, resumed with **resume ()** and stopped using **stop ()**.

Suspend at times create serious troubles as it may happen that if a thread is suspended at a time when it has acquired some resources and now the other threads will be waiting for those resources hence causing the situation of a deadlock, so it is tried to be avoided.

The resume () is also avoided as it is the counterpart pf suspend () and can't be implemented without suspend () .

**Stop()** too sometimes ends up into serious trouble situations so Java 2 came up with a better alternative of run () which will regularly keep a check as to when a thread should resume, suspend or stop its own execution and thus is based on the setting of a flag on each thread which determines the state as running means that the thread should be allowed to run, suspend denoted to be suspended and stop denoting the termination of that particular thread.

## 7 *File Input & Output*

---

### Java Files Input and Output

Java Input Output (IO) is one of the most important topics in Java. Java IO provides many APIs to programmers that he can use to access the files on a filesystem of a computer.

Java has many types of IO classes that can be used to read and write to files, memories, sockets.

Java's I/O libraries are designed in an abstract way that enables you to read from external data sources and write to external targets, regardless of the kind of thing you're writing to or reading from.

You use the same methods to read from a file that you do to read from the console or from a network connection. You use the same methods to write to a file that you do to write to a byte array or a serial port device.

### Stream

#### *What is a Stream?*

A stream is an ordered sequence of bytes of indeterminate length. Input streams move bytes of data into a Java program from some

**JAVA TUTORIAL**

generally external source. Output streams move bytes of data from Java to some generally external target. An input stream may read from a finite source of bytes such as a file or an unlimited source of bytes such as System.in. Similar in an output may write from a finite source of byte such as a file or an unlimited source of byte such as System.out. Most beginners confuse with the term input stream and out stream. They think the other way round. But if you see input stream as a source of input for your program and output stream as a source of output for your program then concept of input and output stream will be much clear to you.

## File Streams

Java includes a particularly rich set of I/O classes in the core API, mostly in the java.io packages. These packages support several different styles of I/O. One distinction is between byte-oriented I/O, which is handled by input and output streams, and character-I/O, which is handled by readers and writers. These all have their place and are appropriate for different needs and use cases.

All classes are basically divided into two types of categories:

1. Byte Oriented I/O
2. Character Oriented I/O

### ***Byte Oriented I/O Classes***

## JAVA TUTORIAL

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from `InputStream` and `OutputStream`.

`java.io.InputStream` (Base Class for `InputStream` Classes)

`java.io.BufferedInputStream`

`java.io.ByteArrayInputStream`

`java.io.DataInputStream`

`java.io.FileInputStream`

`java.io.FilterInputStream`

`java.io.ObjectInputStream`

`java.io.PipedInputStream`

`java.io.PrintStream`

`java.io.PushbackInputStream`

`java.io.SequenceInputStream`

`java.io.OutputStream` (Base Class for `OutputStream` Classes)

`java.io.BufferedOutputStream`

`java.io.ByteArrayOutputStream`

`java.io.DataOutputStream`

`java.io.FileOutputStream`

`java.io.FilterOutputStream`

**JAVA TUTORIAL**

java.io.ObjectOutputStream

java.io.PipedOutputStream

***Character Oriented I/O Classes***

Readers and writers are based on characters, which can have varying widths depending on the character set. For example, ASCII and Latin-1 use 1-byte characters.

UTF-32 uses 4-byte characters. UTF-8 uses characters of varying width (between one and four bytes). Since characters are ultimately composed of bytes, readers take their input from streams.

However, they convert those bytes into chars according to a specified encoding format before passing them along.

Similarly, writers convert chars to bytes according to a specified encoding before writing them onto some underlying stream.

java.io.Reader

java.io.BufferedReader

java.io.CharArrayReader

java.io.FileReader

java.io.FilterReader

java.io.InputStreamReader

**JAVA TUTORIAL**

java.io.LineNumberReader

java.io.PipedReader

java.io.PushbackReader

java.io.StringReader

java.io.Writer

java.io.BufferedWriter

java.io.CharArrayWriter

java.io.FileWriter

java.io.FilterWriter

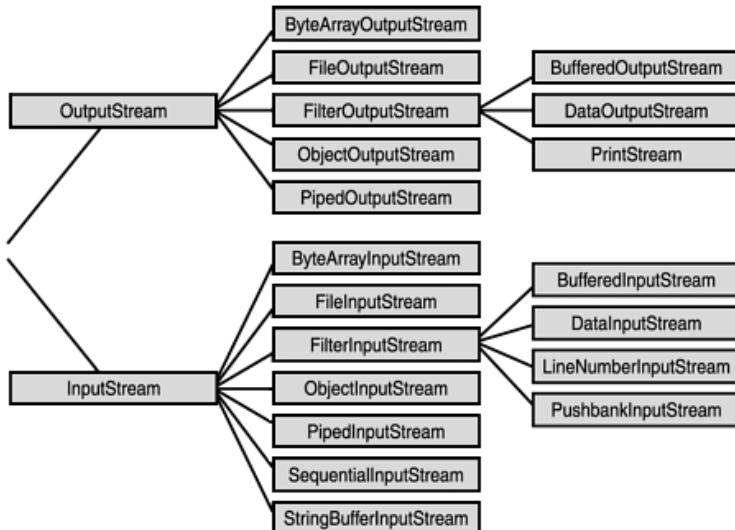
java.io.OutputStreamWriter

java.io.PipedWriter

java.io.PrintWriter

java.io.StringWriter

## JAVA TUTORIAL



### *InputStream Class*

The streaming byte Input of Java is defined by this Abstract Class .all the methods in this class will always throw an IOException whenever any kind of error will occur.

Few methods of this class are as under:

1. **available( )**: This method will return the number of bytes of input which is currently available for reading.
2. **close( )** : this method is used to close the input source. Once the input is closed if we will try to read it again we will get the IOException.

## JAVA TUTORIAL

3. **mark(int numBytes)**: This method places a mark at the current point in the input stream which will remain valid until all the numBytes bytes are read.
4. **markSupported()** : This method since as Boolean will return “true “ if mark( )/reset( ) are supported by the invoking stream.
5. **read( )** : This method returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
6. **reset( )** : This method will reset the input pointer to the previously set mark.

### *OutputStream Class*

This abstract class defines the streaming byte output. All of the methods in this class return a void value and throw an IOException in the case of errors.

Few methods of this class are as under:

1. **close( )** : This method will close Closes the output stream. After the output is closed it will generate an IOException if we will try to perform any operation on it.
2. **flush()** : This method finalizes the output state so that any buffers are cleared. It means that it flushes the output buffers.
3. **write(int b)** : This simple method writes a single byte to an output stream. The parameter isn this method should always be an integer value.

**JAVA TUTORIAL**

4. **write(byte buffer[ ]) :** This method will write a complete array of bytes to an output stream.

***FileInputStream Class***

This class will create an InputStream which can be used to read bytes from a file. the two very basic and common constructors of FileInputStream class are:

1. FileInputStream(String filepath)
2. FileInputStream(File fileObj)

They both can throw a FileNotFoundException if the file is not found. The second constructor should be taken into practice as it helps us to examine the file before attaching it to an input Stream.

```
import java.io.FileInputStream;
import java.io.IOException;

public class FileInputStreamExample {
    public static void main(String[] args) throws
IOException {
    char arr[] = new char[100];
```

**JAVA TUTORIAL**

```
FileInputStream fileStream =  
  
new FileInputStream("myexamplefile.txt");  
  
int data;  
  
for (int i = 0; (data = fileStream.read()) > 0;  
i++) {  
  
    arr[i] = (char) data;  
  
}  
  
System.out.println(arr);  
  
}  
}
```

Be sure that the file called "myexamplesfile.txt" exists in the current directory with some contents in it.

***FileOutputStream Class***

This class creates an OutputStream which is used to write bytes to a file. This is not dependent on the existing file but it itself creates a file before opening it for the output whenever we create the object.

The basic constructors associated to this class are:

1. `FileOutputStream(String filePath):`

**JAVA TUTORIAL**

2. FileOutputStream(File fileObj)
3. FileOutputStream(String filePath, boolean append)
4. FileOutputStream(File fileObj, boolean append)

All of them throw a FileNotFoundException if any error occurs and we get an IOException if we try to access a Read Only File.

```
import java.io.FileOutputStream;  
import java.io.IOException;
```

```
public class FileOutputStreamExample {  
  
    public static void main(String[] args) throws  
IOException {  
  
        String str = "Hello World.";  
  
        FileOutputStream fileOutput =  
            new  
FileOutputStream("myexamplefile.txt");  
  
        fileOutput.write(str.getBytes());  
        fileOutput.flush();  
        fileOutput.close();  
    }  
}
```

**JAVA TUTORIAL**

```
}
```

```
}
```

***ByteArrayOutputStream***

This class is an implementation of an output stream that uses a byte array as the destination. It has two constructors that are shown below:

1. `ByteArrayOutputStream( )` A buffer of 32 bytes is created in this case.
2. `ByteArrayOutputStream(int numBytes)` In this case a buffer will be created which will be equal to the size of numBytes and it can be automatically increased if required.

```
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class ByteArrayOutputStreamExample {
```

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

**JAVA TUTORIAL**

// We need to mock up the scenarion for this example. So we  
// are mocking up the file creation here but in real time  
// it should already be existing.

FileOutputStream filOutputStream = new  
FileOutputStream("myfile.txt");

```
filOutputStream.write("Hello Meshplex.".getBytes());  
filOutputStream.flush();  
filOutputStream.close();
```

byte b[] = new byte[100];

FileInputStream fileInputStream = new  
FileInputStream("myfile.txt");

```
fileInputStream.read(b);
```

ByteArrayOutputStream byteOutputStream = new  
ByteArrayOutputStream();

```
byteOutputStream.write(b);
```

// you can either pass this object to any  
// method that accept OutStream and then

## JAVA TUTORIAL

```
// need to convert into a byte array. Here we
// have used toString() but you can also
// use toByteArray() that will return you the
// byte array.

System.out.println(byteOutputStream.toString());

}
```

### ***FilterInputStream***

These are just the wrappers which surround the input or output streams depending upon the case whether it is an Input or an Output Stream. It has all the methods which are available in the Input Stream.

The constructor for this is,

*FilterInputStream(InputStream is)*

### ***FilterOutputStream***

This class has all the methods similar to the Output Stream .Its constructor is shown below:

`FilterOutputStream(OutputStream os)`

`import java.io.FileNotFoundException;`

**JAVA TUTORIAL**

```
import java.io.FileOutputStream;
import java.io.FilterOutputStream;
import java.io.IOException;

public class FilterInputStreamExample {
    public static void main(String[] args) throws
FileNotFoundException,
IOException {
        byte b[] = "Welcome to Meshplex.".getBytes();
        FilterOutputStream filtereOutputStream =
            new FilterOutputStream(
                new FileOutputStream("newFile.txt"));
        filtereOutputStream.write(b);
        filtereOutputStream.close();
    }
}
```

**JAVA TUTORIAL**

# Buffered Stream

## *BufferedInputStream*

This class provides us with the facility that we can wrap any InputStream into a buffered Stream which will help us to improve its performance to a very higher level.

This class has two constructors which are:

1. `BufferedInputStream(InputStream inputStream)` This will create a buffer with a default buffer size.
2. `BufferedInputStream(InputStream inputStream, int bufSize)` In this case the size of buffer is equal to the variable int bufSize.

This also supports the `mark()` and `reset()` methods. This support is reflected by `BufferedInputStream.markSupported()` returning true.

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.OutputStream;
import java.io.IOException;
```

**JAVA TUTORIAL**

```
public class BufferedInputStreamExample {  
  
    public static void main(String[] args) throws  
FileNotFoundException,  
IOException {  
  
    // Create new File with some data just for this example to work.  
  
    FileOutputStream outFile = new  
FileOutputStream("buffered.txt");  
  
    outFile.write("Hello MeshPlex.".getBytes());  
    outFile.close();  
  
    char buff[] = new char[100];  
    int value;  
  
    BufferedInputStream buffer =  
        new  
BufferedInputStream(new  
FileInputStream("buffered.txt"));  
  
    for (int i = 0; (value = buffer.read()) > 0; i++) {  
        buff[i] = (char) value;  
    }  
}
```

**JAVA TUTORIAL**

```
System.out.println(buff);  
}  
}
```

***BufferedOutputStream***

This stream is similar to any other OutputStream. It has an added exception of flush ( ) method which is used to make sure that the data is written physically on the actual output device. They ensure evry high level of performance with the feature of buffering.

It has two constructors available with it which are as under:

1. `BufferedOutputStream(OutputStream outputStream)` This creates a buffered stream using the buffer of 512 bytes.
2. `BufferedOutputStream(OutputStream outputStream, int bufferSize)` This creates the buffer of the size equal to the variable passed in bufferSize

***PushbackInputStream***

This is implemented on an input stream to provide the access a byte to be read and then returned back to the stream. It works on the mechanism of pushing inside and then seeking it back.

It has the feature through which we can know what is going to come from an Input Stream.

**JAVA TUTORIAL**

This class has the following constructors:

1. PushbackInputStream(InputStream inputStream) This allows the creation of a stream which permits one byte to be returned to the input Stream.
2. PushbackInputStream(InputStream inputStream, int numBytes) This creates a stream that has a pushback buffer that is equal to the length of numBytes long. This allows multiple bytes to be returned to the input stream.

***SequenceInputStream***

This class provides us the feature that we can concatenate the multiple InputStreams. It differs a lot from any other InputStream when there is a comparison between any InputStream and SequenceInputStream.

It contains the following constructors:

1. SequenceInputStream(InputStream first, InputStream second) Here, First and second are the two InputStreams that will be concatenated i.e. combined together to give a new InputStream as the output of this operation.
2. SequenceInputStream(Enumeration streamEnum)

```
import java.io.BufferedReader;  
import java.io.FileInputStream;  
import java.io.FileNotFoundException;
```

## JAVA TUTORIAL

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.SequenceInputStream;
```

```
public class SequenceInputStreamExample {
```

```
    public static void main(String[] args) throws
FileNotFoundException,
IOException {
```

```
        FileOutputStream fileOutput1 = new
FileOutputStream("file1.txt");
        fileOutput1.write("This is my first file.".getBytes());
        fileOutput1.close();
```

```
        FileOutputStream fileOutput2 = new
FileOutputStream("file2.txt");
        fileOutput2.write("This is my second file.".getBytes());
        fileOutput2.close();
```

**JAVA TUTORIAL**

```
FileInputStream fileIn1 = new FileInputStream("file1.txt");
```

```
FileInputStream fileIn2 = new FileInputStream("file2.txt");
```

```
SequenceInputStream sequenceInputStream =
```

```
    new SequenceInputStream(fileIn1, fileIn2);
```

```
BufferedReader reader =
```

```
    new                                                                        BufferedReader(new
```

```
                                                                                InputStreamReader(sequenceInputStream));
```

```
    System.out.println(reader.readLine());
```

```
}
```

```
}
```

***PrintStream***

PrintStream class can be said to be the class which is very much responsible for all type of formatting capabilities which we need to use while working from the System file handle, System.out. It supports both print() and println() methods for all types, including the Object too.

It has the following two constructors:

## JAVA TUTORIAL

1. PrintStream(OutputStream outputStream)
2. PrintStream(OutputStream outputStream, boolean flushOnNewline)

In this flushOnNewline keeps a check as to whether Java will flush the output stream every time we get a newline (\n) character as the output. It will perform the flush operation if it has Boolean value as True” else will not do anything as it gets the value as “false”.

### ***Reader Class***

This class defines the model of streaming character input in Java. This class also gives the IOException whenever there is any error in the Stream.

Few of the methods of this class are explained below in the section:

1. abstract void close( ) : This method will close the input source and once it is closed if we try to read it we will get an IOException as it becomes an error since we are trying to read a closed source.
2. void mark(int numChars) : This method functions for placing a mark at the current point in the input stream that will remain valid until numChars characters are read.
3. boolean markSupported( ) : This method since has Boolean in the strating of writing this method will return the output as

**JAVA TUTORIAL**

either true or false. It returns “True “if mark( )/reset( ) are supported on this stream else returns “false”

4. int read( ) : This method returns an integer representation of the next available character from the invoking input stream. We get -1 as the output if we reach the end of the file.
5. int read(char buffer[ ]) : it tries to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read and gives -1 as value when we reach the end of the file.

***Writer Class***

This class defines the streaming character output. We get a return value as void value and get an IOException in the case of errors in all its methods.

Few of the methods in this class are:

- 1.abstract void close() : This method will close the output stream and will give the IOException. If we try to access the closed stream as it becomes an error to try to access a closed output Stream.
- 2.abstract void flush( ): It finalizes the output state and makes sure that buffers .
- 3.void write(int ch) : This method as defined will write a single character to the invoking output stream.
4. void write(char buffer[ ]) : This method writes a complete array of characters to the invoking output stream.

## JAVA TUTORIAL

### ***FileReader Class***

This class creates a Reader which is used to read the contents of a file. Both the constructors which are defined below can throw the exception FileNotFoundException if file is not found.

The two most common constructors of this class are:

1. FileReader(String filePath)
2. FileReader(File fileObj)

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {

    public static void main(String[] args) throws IOException {
        FileReader fileReader;
        try {
            fileReader = new FileReader("myfile.txt");
            int content ;
        }
    }
}
```

**JAVA TUTORIAL**

```
while((content = fileReader.read()) > 0) {  
  
    System.out.print((char)content);  
  
}  
  
}  
  
catch (FileNotFoundException e) {  
  
    System.out.println("File Does not exists in " +  
        "the current directory. Please create a new " +  
        "file named myfile.txt in the current directory");  
  
}  
  
}  
  
}
```

We use FileReader for reading streams of characters but if you want to read streams of raw bytes then consider using a FileInputStream.

***FileWriter***

This class will create a Writer which will be used to write to a file. All the methods of this class will return the Exception as

## JAVA TUTORIAL

IOException in case of getting errors. When we create a File Writer, it creates the file before opening it when we create the object and it doesn't waste the time in looking for the existence of the particular file.

The most common constructors of this class are

1. FileWriter(String filePath)
2. FileWriter(String filePath, boolean append)
3. FileWriter(File fileObj)
4. FileWriter(File fileObj, boolean append)

```
import java.io.FileWriter;  
import java.io.IOException;
```

```
public class FileWriterExample {  
  
    public static void main(String[] args) throws IOException {  
  
        // the constructor takes two parameters.  
        // 1. Name of the file  
        // 2. Whether to open the file in append mode. In append mode
```

**JAVA TUTORIAL**

```
// contents will be appended instead of getting over written.  
  
FileWriter fileWriter = new FileWriter("myfile.txt", false);  
  
// We cannot write String directly using  
// FileInputStream. But using FileWriter  
// we can.  
  
fileWriter.write("Welcome to Meshplex. " +  
    "It is a great place to learn.");  
  
fileWriter.close();  
  
}  
}
```

**CharArrayReader**

This class is implemented from an input stream which uses a character array as the source.

It has two constructors which are :

1. CharArrayReader(char array[ ])
2. CharArrayReader(char array[ ], int start, int numChars)

## JAVA TUTORIAL

In both these constructors we take array as the input source.

### ***CharArrayWriter***

This class very much like the CharArrayReader implements the output stream that uses an array as the final destination the only difference in both is that it takes the input Stream and it takes the Output Source.

It also has two constructors which are defined below:

1. `CharArrayWriter()` This method will create a buffer with the default size.
2. `CharArrayWriter(int numChars)` This will create a buffer equal to the size of numChars Variable. If required the buffer size will automatically increase if required.

### ***BufferedReader***

The bufferedReader Class like any other buffer improves the performance as it buffers the input.

This class has two constructors which are as under:

1. `BufferedReader(Reader inputStream)` This method will create a buffered character stream using a default buffer size.
2. `BufferedReader(Reader inputStream, int bufferSize)` This method takes the size from the value passed in variable bufferSize.

**JAVA TUTORIAL**

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class BufferedReaderExample {

    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("You typed : " + reader.readLine());
    }
}
```

***PushbackReader***

This class returns one or more characters to the input stream and also helps us to have the idea of what is going to be the next character in the Input Stream.

It also like other classes has few constructors:

1. PushbackReader(Reader inputStream) This method pushes one character at a time in the Input Stream.

## JAVA TUTORIAL

2. PushbackReader(Reader inputStream, int bufSize) This method will have the size of pushback buffer equal to the value which is passes in bufSize variable.

This class has three forms which are mentioned below:

1. void unread(int ch) This pushes back the character passed in ch.
2. void unread(char buffer[ ]) This will return the characters in buffer.
3. void unread(char buffer[ ], int offset, int numChars) This form will push back numChars characters beginning at offset from buffer.

We will get an IOException if we try to access the pushback which is full.

### ***PrintWriter***

This class is basically a character-oriented version of PrintStream. It also supports the formatting of stream in print( ) and println( ) methods

This class has four constructors:

1. PrintWriter(OutputStream outputStream)
2. PrintWriter(OutputStream outputStream, boolean flushOnNewline)

**JAVA TUTORIAL**

3. PrintWriter(Writer outputStream)
4. PrintWriter(Writer outputStream, boolean flushOnNewline)

In these constructors the first and third constructor do not have the feature of automatically flusing the stream while second and fourth one can do it as they both have Boolean flushOnNewline which will either have true or false value and if it is true it will flush else will not.

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;
```

```
public class PrintWriterExample {
    public static void main(String[] args) throws
FileNotFoundException, IOException {
        FileOutputStream fileOutput = new
FileOutputStream("mynewfile.txt");
        PrintWriter writer = new PrintWriter(fileOutput);
```

**JAVA TUTORIAL**

```
writer.println("This is first line of the file.");  
writer.println("This is second line of the file.");  
writer.println("This is third line of the file.");  
writer.println("DVD Player : " + 150.00 + " Audio Player : " +  
100.00);  
writer.flush();  
writer.close();  
fileOutput.close();  
}  
}
```

***InputStreamReader***

This class is a bridge between character oriented I/O classes and byte oriented I/O classes for reading purpose. Using this class we can make Reader classes to read data from InputStream classes.

This class has four constructors:

1. InputStreamReader(InputStream in)
2. InputStreamReader(InputStream in, Charset cs)
3. InputStreamReader(InputStream in, CharsetDecoder dec)
4. InputStreamReader(InputStream in, String charsetName)

**import** java.io.[BufferedReader](#);

**JAVA TUTORIAL**

```
import java.io.IOException;
import java.io.InputStreamReader;

public class BufferedReaderExample {

    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("You typed : " + reader.readLine());
    }
}
```

In the above example BufferedReader class is reading data from InputStreamReader.

***OutputStreamWriter***

This class is a bridge between character oriented I/O classes and byte oriented I/O classes for writing purpose. Using this class we can make Writer classes to write data onto OutputStream classes.

This class has four constructors:

1. OutputStreamWriter(OutputStream out)
2. OutputStreamWriter(OutputStream out, Charset cs)

## JAVA TUTORIAL

3. OutputStreamWriter(OutputStream out, CharsetEncoder enc)
4. OutputStreamWriter(OutputStream out, String charsetName)

```
import java.io.BufferedWriter;  
import java.io.ByteArrayOutputStream;  
import java.io.FileNotFoundException;  
import java.io.IOException;  
import java.io.OutputStreamWriter;
```

```
public class OutputStreamWriterExample {
```

```
    public static void main(String[] args) throws  
FileNotFoundException,  
IOException {
```

```
        String str = "Welcome to MeshPlex. ";
```

```
        ByteArrayOutputStream byteArrayOutputStream =
```

```
            new ByteArrayOutputStream();
```

```
        OutputStreamWriter outputStreamWriter =
```

**JAVA TUTORIAL**

```
new  
OutputStreamWriter(byteArrayOutputStream);  
  
    BufferedWriter bufferedWriter = new  
    BufferedWriter(outputStreamWriter);  
  
    bufferedWriter.write(str);  
  
    bufferedWriter.close();  
  
    System.out.println(byteArrayOutputStream.toString());  
  
}  
  
}
```

## Java Networking

### *What is Network Programming*

Network Programming means different to different people. To some programmers networking programming means low-level programming like reading and writing to network sockets. To others the meaning of network programming is broader and includes the design and programming of distributed applications.

Distributed applications are mainly client server applications that use network server for important services, such as security and database access, are one popular interpretation. The important feature of distributed computing is that it involves two or more computers communicating with each other.

## JAVA TUTORIAL

### ***What is a Network***

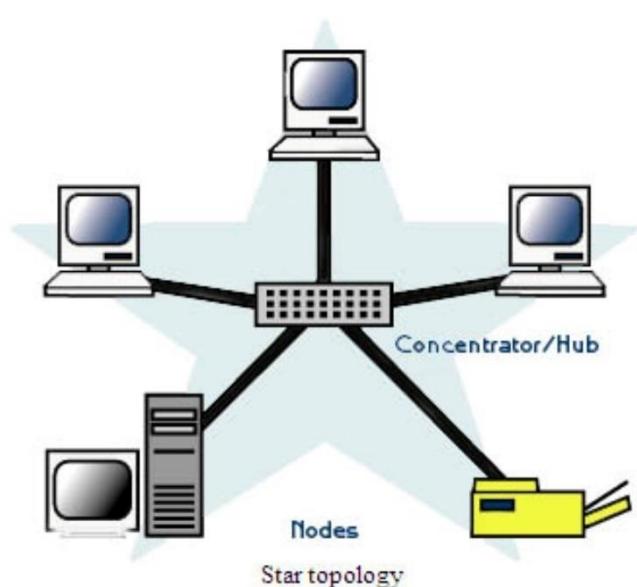
A network is a collection of computers and other devices that can move information around from one device to another. Traditionally, network devices were connected together with wires with shared information converted into electrical signals that were passed across the wires. Later, fiber optics cables were used to convert the information into light signals instead of electrical signals and today, some networks use radio transmitters and receivers and do not use cabling at all. Each device on a network is called a node. These include computers as well as printers, routers, bridges and gateways.

There are lots of different kinds of networks. The most common are LANs (Local Area Networks) and WAN (Wide Area Network). LANs are used to connect computers in a local area such as office, home, shops. WANs connect computers over a wider area, such as across city. Best example for a WAN is Airways Reservation System. Networks are laid out with some common geometric relations. The most common are star, bus and ring topologies.

### **Star topology**

Connects nodes together in smaller circuits, which are then connected to a circuit that runs through the network.

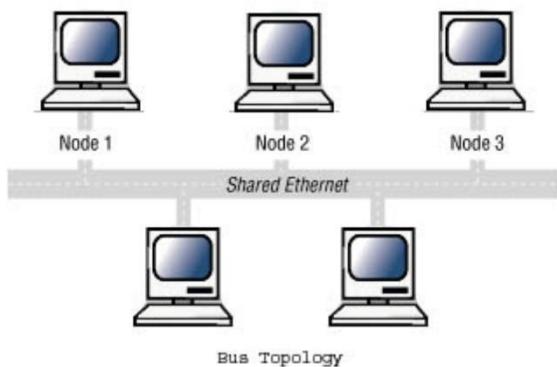
JAVA TUTORIAL



### Bus topology

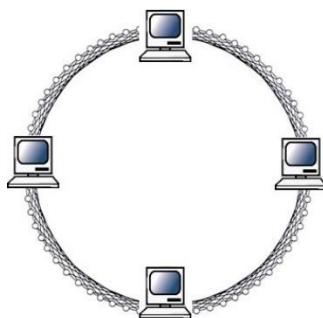
Connects all of the nodes to one circuit that runs through the network like a backbone.

## JAVA TUTORIAL



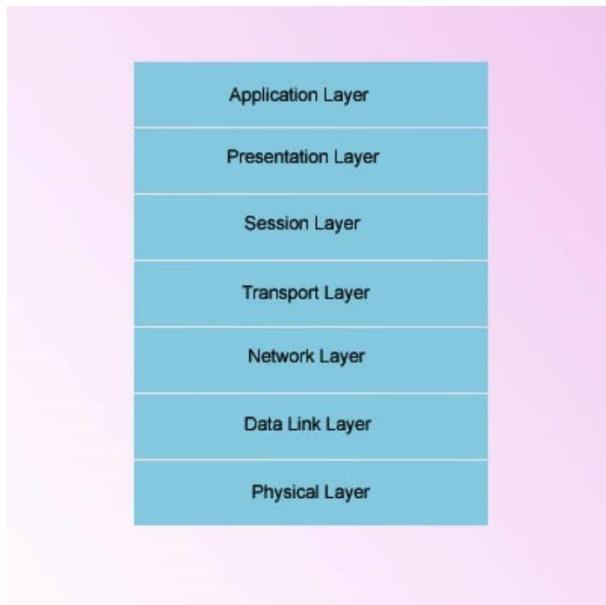
## Ring topology

Connects all of nodes together in one closed circuit



## *Layers of a Network*

## JAVA TUTORIAL



Moving information from one computer to another is a complex set of operations. It involves many different standards and protocols. Engineers solved the complex web of protocols and standards by breaking them down into an abstract set of layers, with each layer building on the services provided by the layers beneath it. The layers of a network only speak to the layer immediately above and below them.

The OSI(Open System Interconnect) Reference Model, shown above defines seven layers that every network stack should provide. This model is defined by the ISO(International Standards Organization)

## JAVA TUTORIAL

Here are some short definitions for these seven layers:

### **Physical Layer**

This layer sits at the lowest level of the OSI model and performs sending and receiving of the raw data through physical mediums like the electrical, optical, and physical components of the network. The physical layer carries the signals for all of the higher layers.

### **Data Link Layer**

The data-link layer provides error-free transfer of data frames from one computer to another through the physical medium. This layer defines the format of the data on the network. The data link layer converts data bits into packets and defines parameters like the largest packet that can be sent. If a packet is corrupted during transit, the data link is responsible for retransmitting the packet.

### **Network Layer**

The Network Layer is responsible for routing those packets received from data link layer. It determines what physical path the data packets should take based on the network conditions, the priority of service, congestions, shortest path and other factors. This layer also breaks packets which are larger than allowed by the underlying network into smaller packets. At the destination point this layer only does the reassembling of those fragmented data packets so that they can be combined into original size.

## JAVA TUTORIAL

### **Transport Layer**

The transport layer ensures that packets are delivered in the order in which they are sent and that there is no loss or duplication. It offers a level of abstraction for the higher layers, shielding them from the low-level details of the network, data link and physical layers.

### **Session Layer**

The session layer starts, maintains and terminates sessions across the network. This layer provides security authentication and synchronization. Synchronization starts check points in the data stream thus improving network performance. At a time session fails, data after the most recent checkpoint will be retransmitted.

### **Presentation Layer**

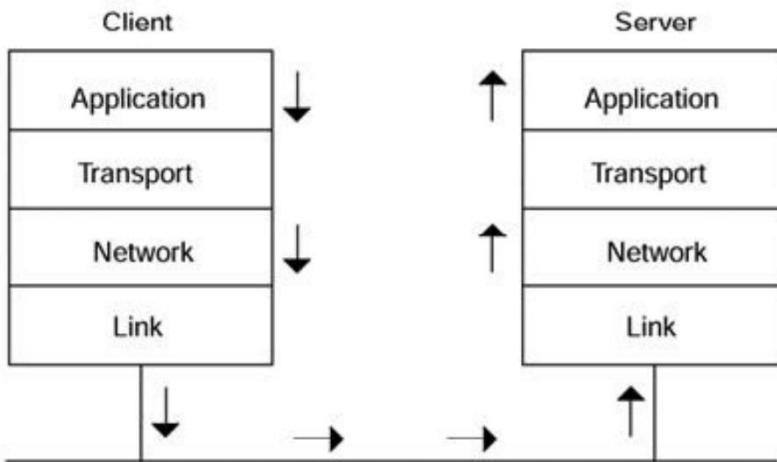
The presentation layer work as the translator of the data for the network. This layer is responsible for the transfer of the data to both the layers that is the next layer above the presentation layer and the one which is below this layer. It performs the character set conversion, encryption and compression of the data.

### **Application Layer**

This layer as defined by its name is responsible for all application services which help to support the user applications. Since it is on

**JAVA TUTORIAL**

the top most hierarchy of network model it works as a user interface for all the applications.

***What are Network Protocols***

The term protocol can be understood in the simplest terms as the standards which have to be followed in the entire process and thus the term Networking Protocols defines the standards or the rules which have to be followed at the time of enabling the network connection and the transfer of the Data across the network.

Thus, we can define Networking Protocols as the rules which govern the syntax, semantics and synchronization of communication. Protocols may be implemented by hardware, software, or a combination of both of them.

## JAVA TUTORIAL

Some of the network Protocols are listed as under:

1. IP (Internet Protocols)
2. TCP (Transmission Control Protocol)
3. UDP (User Datagram Protocol)
4. Ping (Packet InterNetwork Grouping)
5. FTP (File Transfer Protocol)
6. HTTP (Hypertext Transfer Protocol)
7. NNTP (Network News Transfer protocol)
8. SMTP (Simple Mail Transfer Protocol)
9. POP 3 (Post Office Protocol 3)
10. SNMP (Simple Network Management Protocol)

We will discuss each one of them in brief in the section below to have an understanding about each one of them:

1. IP (Internet Protocols): Internet Protocols are very much responsible for sending or transferring the data from one computer to another on the network. The data which has to travel from one computer to another is divided into a number of divisions which are called as “Packets”. Every packet is first passed through a gateway computer which understands a part of the internet. These packets travel all over the network carrying a unique address of both the sender and the receiver which will help the data packet to finally reach the correct destination Computer. These Unique addresses are called the IP address. Both the computers i.e. the sender and the receiver are called the hosts and they both have a 32 bit logical IP Address.

## JAVA TUTORIAL

This protocol is responsible for the successful transfer of data packets from one host to another but it is very much possible the packets do not reach in the same sequence it should reach and to make sure that all the packets reach in the right sequence we have another Protocol which takes care of this. This protocol is called as TCP (Transmission Control Protocol); we will talk about it below in the section.

This protocol does not make any continuous connection between the sender and the receiver and that's why it is termed as connectionless protocol and each packet of data behaves more like an independent unit of the data which should reach the correct destination.

2. TCP (Transmission Control Protocol): Transmission Transfer Protocol works closely with the internet protocol as it handles the responsibility of the successful delivery of the data packets in the correct sequence which are traveling from the receiver to the sender. The complete message is divided into various segments called the packets which help to transfer the complete data or the message successfully over the network.

This Protocol is responsible for ensuring that a message is divided into the packets whose transmission is controlled by the IP and then they are rearranged back in the same order so as to make a meaningful message reach at the destination which is the other end of the transmission process.

## JAVA TUTORIAL

Unlike IP, TCP is called as a connection-oriented protocol because it provides a virtual connection service with the help of sockets and ports. This also means that a connection is established and maintained until the complete message is successfully exchanged between the two hosts.

3. UDP (User Datagram Protocol): User Datagram Protocol provides a very limited and restricted service when the data or the messages are exchanged between the two hosts which implement the Internet Protocol. This protocol sits in the 4th Layer which is the transport Layer. It also uses the internet Protocol to send or transfer a data unit or segment which is called as Datagram from one computer to another.

Since this protocol works like the Internet Protocol it doesn't have any feature which assures the transfer of the datagrams in the correct sequence to the destination Computer this protocol is advised to be implemented only when the data size is very small else it can create trouble at the time of transferring a big data or message from one computer to another. It doesn't provide any error correction or retransmission feature so if any data segment is lost or corrupted in the transmission process the complete data packet is discarded and rejected. So it can work successfully only in case of small independent packets of information.

4. Ping (Packet InterNetwork Grouping): Ping is a very common terminology in the computer world which means that when we

## JAVA TUTORIAL

ping a host on the network, we come to know whether that host is active or inactive or whether there is any delay in the communication between the sender and the receiver.

Ping Packet InterNetwork Grouping is a routing protocol which helps to handle network problems like network congestion. It uses the ICMP, Internet Control Message Protocol messages which try to get the response from the remote host by sending an echo request packet.

One very important point to remember in case of Ping is that it cannot be implemented in Java as Java doesn't support ICMP.

5. FTP (File Transfer Protocol): File Transfer protocol is a very familiar and old protocol. Rather it can be said to be one of the oldest TCP/IP protocols and its term defines its purpose is to transfer the text and binary files successfully between two hosts on the network.

It is mostly used to send the Web page files from the sender to the computer which acts as a server for everyone on the Internet. And also used for the purpose of downloading the files and other programs to our computer from other servers. We can implement FTP with the help of a very simple command line interface and also with the help of GUI, graphical user Interface. It can use the password-controlled access and public access too. When it is password Controlled then it is controlled with the help of account

## JAVA TUTORIAL

and password and if it is made public then no account or password is required.

6. HTTP (Hypertext Transfer Protocol): Whenever we need to transfer the various types of files on the World Wide Web we implement the rules and guidelines provided by Hypertext Transfer Protocol. This protocol works on the top of TCP/IP Protocols which acts as the base of the protocols on the internet.

An HTTP protocol is used for communicating between the Web Browsers and the Web Browsers and it comes into picture immediately when we open any Web Browser.

The basic functionality in this process is that the web browser opens a connection and makes a request to the server, once the server sends back the requested information the connection is closed. Thus, it is termed as a stateless protocol.

7. NNTP (Network News Transfer protocol): As the term defines itself this protocol is used by the USENET Internet news System. It is basically implemented by the computer clients and servers which try to manage the notes published on Usenet Newsgroups.

These servers try to efficiently manage the global network of all the collected Usenet newsgroups and include the server at the Internet access provider. An NNTP client is included as part of any Web Browser like Netscape, Internet Explorer etc and we also

## JAVA TUTORIAL

have the flexibility to use a separate client program called a newsreader.

8. SMTP (Simple Mail Transfer Protocol) SMTP is used to send and receive E-Mails and is a TCP/IP Protocol. It is used in combination either with POP 3 or IMAP which helps the user to save all its messages in a mailbox and then download them later and this feature is not available with SMTP and thus used in combination with POP 3 or IMAP. SMTP.

Commonly, SMTP is used for sending messages from a client to a server and POP 3 or IMAP are used for receiving the E-Mails. Here the client establishes a connection with the SMTP server through a request or response dialogue and then the client transmits the mail addresses of the recipients for a message. If the given request is processed by the server and it accepts the mail address the message is successfully transmitted to the recipient.

9. POP 3 (Post Office Protocol 3): This protocol is said to be the latest version of a standard protocol which is used to receive the Emails. This is commonly used with the SMTP. SMTP manages sending the Emails and POP 3 serves the purpose of receiving the Emails for the recipients. The recipient can any time check his/her mailbox and download any mail from the mail box. It deletes the mail on the server once the mail gets downloaded by the receiver. This standard protocol is built into all most all popular e-mail products, such as Outlook Express and Microsoft Internet Explorer.

## JAVA TUTORIAL

browsers. An alternative protocol is Internet Message Access Protocol; IMAP which functions in a very similar way except that it retains e-mails on the server and it can also be called as a remote file server for organizing them in folders on the server.

10. SNMP (Simple Network Management Protocol): This protocol serves the purpose of controlling all the devices like bridges, routers and hubs etc which are connected to a network. All the resources which are monitored by SNMP are called as “managed object”.

It is also a connectionless protocol and quite simple in its functionality and thus it is recommended for very simple management of network components.

### ***Network Models***

The networking strategies of modern world are fully based on two types of models which are:

1. Client /Server Model
2. Peer-to-peer Model.

We will now talk about both of them to have an understanding on each in the section below:

1. Client/Server Model: The relationship between two computer programs in which they carry the relationship of a Client and a

## JAVA TUTORIAL

server represents the client/Server Model of networking. In this one program which is basically denoted as client, makes a request for any service from the other computer program which is called as server and the server in return to the request made by the client fulfills its request by performing that service. This model makes it very easy for the two computer programs to be connected to each other although they might be placed at two very different far off locations. Both the server and the client run their individual processes and the server processes can be executed either on the client machine or it can be easily executed on the remote machine.

It has successfully made an important place in the networking world and that's why it is in today's time is basically implemented in all the major business applications. The most common example where we see this model running very effectively and successfully is the operations which are performed on day-to-day basis in any Bank. Here all the communication is been done between two computer programs in which one server as a client and the other as a server.

The two very broad areas where we can see the client server running very successfully are Custom Business applications and Internet Applications.

In Java, we have the option that we can choose and decide how to split and distribute the work between the client and the server processes so as to make the entire application very efficient and here we have few choices with respect to the architecture which are:

## JAVA TUTORIAL

1. Two-Tier Architecture
2. Three-Tier Architecture
3. Multi –Tier Architecture

Now we will briefly take a look on all the three architectures:

1. Two-Tier Architecture: The two- Tier Architecture as the term illustrates is based on the very simple and common architecture wherein we have a “fat” server and a “thin” client and all the major processes are been brought under the dominance of the server.

The biggest positive aspect of the two-Tier architecture is that it is very simple and easy to use because all the TopLink features which the two-tier architecture provides are present in a single session type and the biggest drawback present in this architecture is that each client of it needs its own individual database session which provides the TopLink support for the two-Tier architecture. Here mainly a Java client connects directly to the database with the help of TopLink.

The Two-tier architecture is not that common as the three-tier architecture model.

2. Three-Tier Architecture: The three Tier architecture as the name defines divides the entire application into three components User Interface Services Tier, Business Rule Services Tier, Data Persistence Services Tier.

## JAVA TUTORIAL

In this type of model, the client application behaves as a very thin client. It as its term mentions, serves the purpose of providing only the user interface and it has nothing to do with the business logic of the particular application. This responsibility is taken care by the Application Servers which communicates directly with the database servers and since this has been isolated by the user interface and the data persistence, they can easily be put in the business rule components and these components can be installed on any number of application servers which makes it more effective than the two-Tier architecture model.

This mechanism not only makes it more compatible it also increases the performance of the application servers and whenever we do any kind of changes to the business rules, we can get it modified on only few application servers instead doing these changes on many workstations thus saving huge amount of time.

This architecture is suggested to be implemented in the case of Web Application. Difference between the two-Tier and Three-Tier architecture: The basic difference between the two-tier and three-tier architecture is that first one is a client/server architecture, in which the client requests to perform a task to the server and gets the response from the server in the form of execution of that task where as a three-tier or a multi-tier architecture constitutes of a client, server and database. Where the client sends the request to the server and the server then sends the request to the database and the database sends information/data required back to the server and the server finally sends the information to the client for which it made the request from the server.

## JAVA TUTORIAL

3. Multi-Tier architecture: This architecture is also termed as the n-tier architecture. This architecture unlike the three-tier architecture further bifurcates the business rule services into two segments. One is for supporting the user interface and the other to manipulate and play around the data.

It can be categorized into four tiers. First the User Interface Services Tier, second is the UI-Oriented Services Tier, third the Data-Oriented Business Rule Services tier and the last one the Data Persistence Services Tier.

The basic difference between the three-Tier model and the multi-tier model is that this model further divides the business services tier present in the three-Tier architecture to two more divisions making it more flexible to be implemented. In this model we can have more processes added to the application servers and to increase the performance we can also add few more servers if required.

### ***Multicasting***

Multicasting can be defined as the process through which we can send a single message to n number of recipients at one point of time. Thus, it saves a lot of time. It is supported through the wireless data networks. The best example to properly understand this concept is teleconferencing and videoconferencing. But this also doesn't provide any kind of assurance as so deliver the data or message successfully to all the recipients. Many a time people mix multicasting with broadcasting but they vary from each other

## JAVA TUTORIAL

and the biggest difference between multicasting and broadcasting is that multicasting is used when we have to send a message to a group of selected recipients whereas broadcasting is used when we are sending a message to everyone who is connected to a network. Multicasting plays an effective role in the areas like distance learning, digital video libraries, online collaboration tools, and other types of advanced applications which are important to research and education.

All the java programs communicate with a help of a programming abstraction which is called a socket. It basically denotes the end point of any network communication. The benefit of having a file abstraction is that all the details related to the storage of file, their location, how they are read and written is all hidden under this abstraction and this as a result gives the programmer a very easy and a simple functional programming interface.

Most applications do not care how data is actually transmitted in the network. Applications identify the address of the peer entity and then use the sockets interface to read and write data from and to their peer. Sockets combine the implementation of network and transport layer protocols providing applications with a simple read/write interface. Java provides two types of sockets:

1. Server Sockets
2. Sockets

### *Server Sockets*

**JAVA TUTORIAL**

Example for ServerSocket

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;

public class ServerSocketExample {

    public static void main(String[] args) throws IOException {
        InetAddress address = InetAddress.getByName("127.0.0.1");

        ServerSocket server = new ServerSocket(5464, 0, address);

        System.out.println(server.getInetAddress().getHostAddress());
        System.out.println("Listening");
        Socket clientSocket = server.accept();
```

## JAVA TUTORIAL

```
System.out.println("Connection Accepted...");  
BufferedReader reader =  
new BufferedReader(new InputStreamReader(  
clientSocket.getInputStream()));  
System.out.println(reader.readLine());  
}  
}
```

### *Sockets*

Example for Socket

```
import java.io.IOException;  
import java.io.PrintWriter;  
import java.net.InetAddress;  
import java.net.Socket;  
import java.net.UnknownHostException;  
  
public class SocketExample {  
  
    public static void main(String[] args) throws IOException {
```

## JAVA TUTORIAL

```
Socket socket = null;  
PrintWriter printWriter = null;  
  
InetAddress address = InetAddress.getByName("127.0.0.1");  
  
try {  
  
    socket = new Socket(address, 5464);  
    printWriter = new PrintWriter(socket.getOutputStream(),  
        true);  
    printWriter.println("Hello MeshPlex");  
    printWriter.flush();  
}  
  
catch (UnknownHostException e) {  
    System.out.println("Don't know about host: 127.0.0.1");  
}  
  
catch (IOException e) {  
    System.out.println("Couldn't get I/O for "  
        + "the connection to: 127.0.0.1");  
}
```

**JAVA TUTORIAL**

```
    }  
  
    printWriter.close();  
  
    socket.close();  
  
}  
  
}
```

### URL Connection class

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for URL objects that are using the HTTP protocol.

The **Java URLConnection** class represents a communication link between the URL and the application. This class can be used to read and write data to the specified resource referred by the URL.

To get the object of URLConnection class

The openConnection() method of URL class returns the object of URLConnection class. Syntax:

**JAVA TUTORIAL**

1. **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. The **getInputStream()** method returns all the data of the specified URL in the stream that can be read and displayed.

### ***Summary***

#### Stream Classes

<b>Stream Class</b>	<b>Meaning</b>
<b>BufferedReader</b>	Buffered input character stream
<b>BufferedWriter</b>	Buffered output character stream
<b>CharArrayReader</b>	Input stream that reads from a character array
<b>CharArrayWriter</b>	Output stream that writes to a character array
<b>FileReader</b>	Input stream that reads from a file
<b>FileWriter</b>	Output stream that writes to a file
<b>StringReader</b>	Input stream that reads from a string
<b>StringWriter</b>	Output stream that writes to a string

#### Methods in Console

**JAVA TUTORIAL**

Method	Description
<b>Reader reader()</b>	Retrieve the reader object associated with the console
<b>String readLine()</b>	Read a single line of text from the console.
<b>String readLine(String fmt, Object... args)</b>	Provides a formatted prompt and reads the single line of text from the console.
<b>Console format(String fmt, Object... args)</b>	Write a formatted string to the console output stream.
<b>Console printf(String format, Object... args)</b>	Write a string to the console output stream.
<b>PrintWriter writer()</b>	Retrieve the PrintWriter object associated with the console.
<b>void flush()</b>	Flushes the console.

**Writer class**

Modifier and Type	Method	Description
<b>Writer</b>	<b>append(char c)</b>	Appends the specified character to this writer.
<b>Writer</b>	<b>append(CharSequence csq)</b>	Appends the specified character sequence to this writer
<b>Writer</b>	<b>append(CharSequence csq, int start, int end)</b>	Appends a subsequence of specified character sequence to this writer.
<b>abstract void</b>	<b>close()</b>	Closes the stream, flushing it first.
<b>abstract void</b>	<b>flush()</b>	Flushes the stream.

**Reader Class**

## JAVA TUTORIAL

Modifier and Type	Method	Description
<b>abstract void</b>	<b>close()</b>	Close the stream and releases any system resource associated with it
<b>void</b>	<b>mark(int readAheadLimit)</b>	Identify the present position in the stream
<b>boolean</b>	<b>markSupported()</b>	Says stream supports the mark() operation
<b>int</b>	<b>read()</b>	Reads a single character
<b>int</b>	<b>read(char[] cbuf)</b>	Reads characters into an array
<b>abstract int</b>	<b>read(char[] cbuf, int off, int len)</b>	Reads characters into a portion of an array

## String Reader

Method	Description
<b>int read()</b>	It is used to read a single character.
<b>int read(char[] cbuf, int off, int len)</b>	It is used to read a character into a portion of an array.
<b>boolean ready()</b>	It is used to tell whether the stream is ready to be read.
<b>boolean markSupported()</b>	It is used to tell whether the stream support mark() operation.
<b>long skip(long ns)</b>	It is used to skip the specified number of character in a stream

## Piped Reader

**JAVA TUTORIAL**

<b>Modifier and Type</b>	<b>Method</b>	<b>Method</b>
<b>void</b>	<b>close()</b>	Close the piped stream and releases system resources associated with the stream.
<b>void</b>	<b>connect(PipedWriter src)</b>	Makes piped reader to be connected to the piped writer src.
<b>int</b>	<b>read()</b>	Reads the next character of data from this piped stream.
<b>int</b>	<b>read(char[] cbuf, int off, int len)</b>	Reads up to len characters of data from this piped stream into an array of characters.
<b>boolean</b>	<b>ready()</b>	Says whether this stream is ready to be read.

**Piped Writer**

<b>Modifier and Type</b>	<b>Method</b>	<b>Method</b>
<b>void</b>	<b>close()</b>	It closes this piped output stream and releases any system resources associated with this stream.
<b>void</b>	<b>connect(PipedReader snk)</b>	It connects this piped writer to a receiver.
<b>void</b>	<b>flush()</b>	It flushes this output stream and forces any buffered output characters to be written out.
<b>void</b>	<b>write(char[] cbuf, int off, int len)</b>	It writes len characters from the specified character array starting at offset off to this piped output stream.

**JAVA TUTORIAL**

## **8 Java AWT**

---

### **Basics of AWT**

Graphical User Interface (GUI) offers user interaction via some graphical components. For example our underlying Operating System also offers GUI via window, frame, Panel, Button, Textfield, TextArea, Listbox, Combobox, Label, Checkbox etc. These all are known as components. Using these components we can create an interactive user interface for an application.

GUI provides result to end user in response to raised events. GUI is entirely based events. For example, clicking over a button, closing a window, opening a window, typing something in a textarea etc. These activities are known as events. GUI makes it easier for the end user to use an application. It also makes them interesting.

There are two varieties of applets based on Applet. The first are those based directly on the Applet class. These applets use the Abstract Window Toolkit (AWT) to provide the graphical user interface (or use no GUI at all). This style of applet has been available since Java was first created. The second type of applets are those based on the Swing class JApplet, which inherits Applet. The Abstract Window Toolkit (AWT) was Java's first GUI framework, and it has been part of Java since version 1.0. It contains numerous classes and methods that allow you to create windows and simple controls. It is important to state at the outset that you will seldom create GUIs based

## JAVA TUTORIAL

solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. Despite this fact, the AWT remains an important part of Java.

An understanding of the AWT is still important because the AWT underpins Swing, with many AWT classes being used either directly or indirectly by Swing. As a result, a solid knowledge of the AWT is still required to use Swing effectively.

**Java AWT** (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

Java AWT components are platform-dependent i.e., components are displayed according to the view of operating system. AWT is heavyweight i.e., its components are using the resources of OS.

The `java.awt` package provides classes for AWT api such as

- `TextField`,
- `Label`,
- `TextArea`,
- `RadioButton`,
- `CheckBox`,
- `Choice`,
- `List` etc.

### Components and containers

## JAVA TUTORIAL

All the elements like buttons, text fields, scrollbars etc are known as components. In AWT we have classes for each component as shown in the above diagram. To have everything placed on a screen to a particular position, we have to add them to a container. A container is like a screen wherein we are placing components like buttons, text fields, checkbox etc. In short, a container contains and controls the layout of components. A container itself is a component (shown in the above hierarchy diagram) thus we can add a container inside container.

### **Types of Containers:**

As explained above, a container is a place wherein we add components like text field, button, checkbox etc. There are four types of containers available in AWT: Window, Frame, Dialog and Panel. As shown in the hierarchy diagram above, Frame and Dialog are subclasses of Window class.

**Window:** An instance of the Window class has no border and no title.

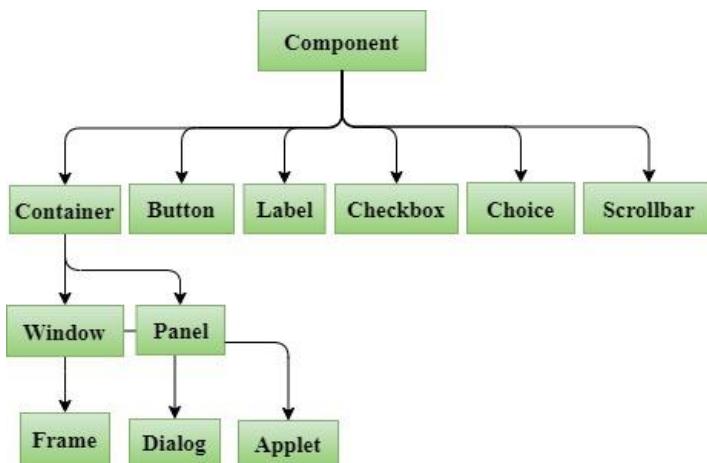
**Dialog:** Dialog class has border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.

**Panel:** Panel does not contain title bar, menu bar or border. It

## JAVA TUTORIAL

is a generic container for holding components. An instance of the Panel class provides a container to which to add components.

**Frame:** A frame has title, border and menu bars. It can contain several components like buttons, text fields, scrollbars etc. This is most widely used container while developing an application in AWT.



## Event Handling

Since Frame is a subclass of Component, it inherits all the capabilities defined by Component. This means that you can use and manage a frame window just like you manage an applet's main window, as described earlier in this book. For

## JAVA TUTORIAL

example, you can override `paint()` to display output, call `repaint()` when you need to restore the window, and add event handlers. Whenever an event occurs in a window, the event handlers defined by that window will be called. Each window handles its own events.

Change in the state of an object is known as event i.e., event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

### *Types of Event*

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or

## JAVA TUTORIAL

software failure, timer expires, an operation completion is the example of background events.

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Following is the declaration for **java.util.EventObject** class:

```
public class EventObject extends Object implements  
Serializable
```

Class constructors

## JAVA TUTORIAL

**EventObject(Object source)** - Constructs a prototypical Event.

Class Methods

**Object getSource()** - The object on which the Event initially occurred.

**String toString()** - Returns a String representation of this EventObject.

1. AWTEvent - It is the root event class for all AWT events. This class and its subclasses supercede the original java.awt.Event class.
2. ActionEvent - The ActionEvent is generated when button is clicked or the item of a list is double clicked.
3. InputEvent - The InputEvent class is root event class for all component-level input events.
4. KeyEvent - On entering the character the Key event is generated.
5. MouseEvent - This event indicates a mouse action occurred in a component.
6. TextEvent - The object of this class represents the text events.

**JAVA TUTORIAL**

7. WindowEvent - The object of this class represents the change in state of a window.
8. AdjustmentEvent - The object of this class represents the adjustment event emitted by Adjustable objects.
9. ComponentEvent - The object of this class represents the change in state of a window.
10. ContainerEvent - The object of this class represents the change in state of a window.
11. MouseMotionEvent - The object of this class represents the change in state of a window.
12. PaintEvent - The object of this class represents the change in state of a window.

JAVA TUTORIAL

## AWT Controls

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Editing

These controls are subclasses of Component. Although this is not a particularly rich set of controls, it is sufficient for simple applications.

Every user interface considers the following three main aspects:

- **UI elements:** These are the core visual elements the user eventually sees and interacts with. GWT provides a huge list of widely used and common elements varying from basic to complex.
- **Layouts:** They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface).

## JAVA TUTORIAL

**Behavior:** These are events which occur when the user interacts with UI elements.

### 1. Label

A Label object is a component for placing text in a container.

### 2. Button

This class creates a labeled button.

### 3. Check Box

A check box is a graphical component that can be in either an **on** (true) or **off** (false) state.

### 4. Check Box Group

The CheckboxGroup class is used to group the set of checkbox.

### 5. List

The List component presents the user with a scrolling list of text items.

### 6. Text Field

A TextField object is a text component that allows for the editing of a single line of text.

### 7. Text Area

A TextArea object is a text component that allows for the editing of a multiple lines of text.

### 8. Choice

A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.

### 9. Canvas

**JAVA TUTORIAL**

A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.

**10. Image**

An Image control is superclass for all image classes representing graphical images.

**11. Scroll Bar**

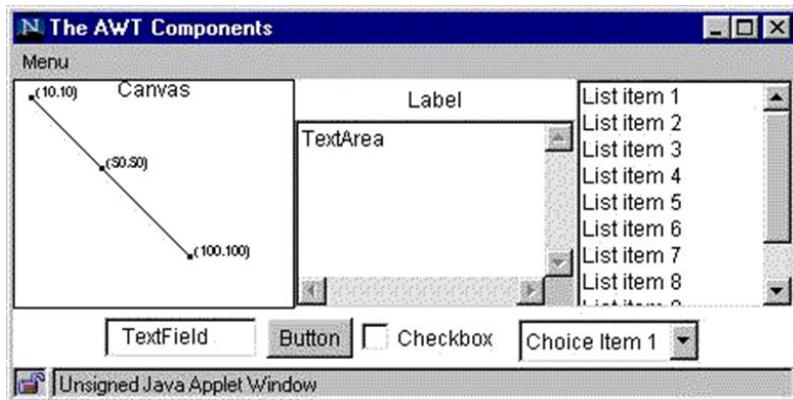
A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

**12. Dialog**

A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.

**13. File Dialog**

A FileDialog control represents a dialog window from which the user can select a file.



## JAVA TUTORIAL

# Design programming concept

Design-oriented programming is a way to author computer applications using a combination of text, graphics, and style elements in a unified code-space. The goal is to improve the experience of program writing for software developers, boost accessibility, and reduce eye-strain. Good design helps computer programmers to quickly locate sections of code using visual cues typically found in documents and web page authoring.

User interface design and graphical user interface builder research are the conceptual precursors to design-oriented programming languages. The former focus on the software experience for end users of the software application and separate editing of the user interface from the code-space. The important distinction is that design-oriented programming involves user experience of programmers themselves and fully merges all elements into a single unified code-space.

Design principles are generalized pieces of advice or proven good coding practices that are used as rules of thumb when making design choices.

They're a similar concept to design patterns, the main difference being that design principles are more abstract and generalized. They are high-level pieces of advice, often applicable to many different programming languages or even different paradigms.

## JAVA TUTORIAL

Design patterns are also abstractions or generalized good practices, but they provide much more concrete and practical low-level advice, and are related to entire classes of problems rather than just generalized coding practices.

Software design principles represent a set of guidelines that helps us to avoid having a bad design. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices". According to Robert Martin there are 3 important characteristics of a bad design that should be avoided:

Rigidity - It is hard to change because every change affects too many other parts of the system.

Fragility - When you make a change, unexpected parts of the system break.

Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

### ***Open Close Principle***

Software entities like classes, modules and functions should be open for extension but closed for modifications.

OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don't have to change the class but to extend it. The same principle can be applied for modules, packages, libraries. If you have a library containing a set of classes there

## JAVA TUTORIAL

are many reasons for which you will prefer to extend it without changing the code that was already written (backward compatibility, regression testing). This is why we have to make sure our modules follow Open Closed Principle.

When referring to the classes Open Close Principle can be ensured by use of Abstract Classes and concrete classes for implementing their behavior. This will enforce having Concrete Classes extending Abstract Classes instead of changing them. Some particular cases of this are Template Pattern and Strategy Pattern.

### *Dependency Inversion Principle*

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high-level classes and low-level classes. Furthermore, it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.

Dependency Inversion or Inversion of Control are better known terms referring to the way in which the dependencies are realized. In the classical way when a software module (class, framework) needs some other module, it initializes and holds a

## JAVA TUTORIAL

direct reference to it. This will make the 2 modules tight coupled. In order to decouple them the first module will provide a hook (a property, parameter) and an external module controlling the dependencies will inject the reference to the second one.

By applying the Dependency Inversion, the modules can be easily changed by other modules just changing the dependency module. Factories and Abstract Factories can be used as dependency frameworks, but there are specialized frameworks for that, known as Inversion of Control Container.

### ***Interface Segregation Principle***

Clients should not be forced to depend upon interfaces that they don't use.

This principle teaches us to take care how we write our interfaces. When we write our interfaces, we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well. For example, if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?

As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. We should avoid them.

### ***Single Responsibility Principle***

## JAVA TUTORIAL

A class should have only one reason to change.

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change, we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Single Responsibility Principle was introduced Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

### *Liskov's Substitution Principle*

Derived types must be completely substitutable for their base types.

This principle is just an extension of the Open Close Principle in terms of behavior meaning that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.

Liskov's Substitution Principle was introduced by Barbara Liskov in a 1987 Conference on Object Oriented Programming Systems Languages and Applications, in Data abstraction and hierarchy.

**JAVA TUTORIAL**

## Listeners

**Java AWT Listeners** are a group of interfaces from **java.awt.event** package. Listeners are capable to handle the events generated by the components like button, choice, frame etc. These listeners are implemented to the class which requires handling of events.

```
public class ButtonDemo1 extends Frame implements  
ActionListener
```

The class **ButtonDemo1** implements **ActionListener** as **ButtonDemo1** includes some buttons which require event handling. The button events (known as action events) are handled by **ActionListener**.

Even though, some listeners handle the events of a few components, generally every component event is handled by a separate listener. For example, the **ActionListener** handles the events of **Button**, **TextField**, **List** and **Menu**. But these types are very rare.

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event**. For

**JAVA TUTORIAL**

example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

***Delegation Event Model***

1. Implement the appropriate interface in the listener so that it can receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

<b>Interfaces</b>	
ActionListener	KeyListener
AdjustmentListener	MouseListener
ComponentListener	MouseMotionListener
ContainerListener	MouseWheelListener
FocusListener	TextListener
ItemListener	WindowFocusListener
WindowListener	

The delegation event model has two parts: sources and listeners. As it relates to this chapter, listeners are created by implementing one or more of the interfaces defined by the java.awt.event package. When an event occurs, the event source

**JAVA TUTORIAL**

invokes the appropriate method defined by the listener and provides an event object as its argument.

***Listener Interfaces:***

The ActionListener Interface This interface defines the actionPerformed( ) method that is invoked when an action event occurs. Its general form is shown here:

*void actionPerformed(ActionEvent ae)*

The AdjustmentListener Interface This interface defines the adjustmentValueChanged( ) method that is invoked when an adjustment event occurs. Its general form is shown here: *void adjustmentValueChanged(AdjustmentEvent ae)*

The ComponentListener Interface This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here: *void componentResized(ComponentEvent ce)*

*void componentMoved(ComponentEvent ce)*

*void componentShown(ComponentEvent ce)*

*void componentHidden(ComponentEvent ce)*

The ContainerListener Interface This interface contains two methods. When a component is added to a container, componentAdded( ) is invoked. When a component is removed from a container, componentRemoved( ) is invoked. Their general forms are shown here:

**JAVA TUTORIAL**

*void componentAdded(ContainerEvent ce)*

*void componentRemoved(ContainerEvent ce)*

The FocusListener Interface This interface defines two methods. When a component obtains keyboard focus, *focusGained( )* is invoked. When a component loses keyboard focus, *focusLost( )* is called. Their general forms are shown here:

*void focusGained(FocusEvent fe)*

*void focusLost(FocusEvent fe)*

The ItemListener Interface This interface defines the *itemStateChanged( )* method that is invoked when the state of an item changes. Its general form is shown here:

*void itemStateChanged(ItemEvent ie)*

The KeyListener Interface This interface defines three methods. The *keyPressed( )* and *keyReleased( )* methods are invoked when a key is pressed and released, respectively. The *keyTyped( )* method is invoked when a character has been entered. For example, if a user presses and releases the a key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the home key, two key events are generated in sequence: key pressed and released. The general forms of these methods are shown here:

*void keyPressed(KeyEvent ke)*

**JAVA TUTORIAL***void keyReleased(KeyEvent ke)**void keyTyped(KeyEvent ke)*

The **MouseListener** Interface This interface defines five methods. If the mouse is pressed and released at the same point, `mouseClicked( )` is invoked. When the mouse enters a component, the `mouseEntered( )` method is called. When it leaves, `mouseExited( )` is called. The `mousePressed( )` and `mouseReleased( )` methods are invoked when the mouse is pressed and released, respectively. The general forms of these methods are shown here:

*void mouseClicked(MouseEvent me)**void mouseEntered(MouseEvent me)**void mouseExited(MouseEvent me)**void mousePressed(MouseEvent me)**void mouseReleased(MouseEvent me)*

The **MouseMotionListener** Interface This interface defines two methods. The `mouseDragged( )` method is called multiple times as the mouse is dragged. The `mouseMoved( )` method is called multiple times as the mouse is moved. Their general forms are shown here:

*void mouseDragged(MouseEvent me)**void mouseMoved(MouseEvent me)*

## JAVA TUTORIAL

The MouseWheelListener Interface This interface defines the mouseWheelMoved( ) method that is invoked when the mouse wheel is moved. Its general form is shown here:

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

The TextListener Interface This interface defines the textValueChanged( ) method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

```
void textValueChanged(TextEvent te)
```

The WindowFocusListener Interface This interface defines two methods: windowGainedFocus( ) and windowLostFocus( ). These are called when a window gains or loses input focus. Their general forms are shown here:

```
void windowGainedFocus(WindowEvent we)
```

```
void windowLostFocus(WindowEvent we)
```

The WindowListener Interface This interface defines seven methods. The windowActivated( ) and windowDeactivated( ) methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified( ) method is called. When a window is deiconified, the windowDeiconified( ) method is called. When a window is opened or closed, the windowOpened( ) or windowClosed( ) methods are called, respectively. The windowClosing( ) method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)
```

**JAVA TUTORIAL**

```
void windowClosed(WindowEvent we)  
void windowClosing(WindowEvent we)  
void windowDeactivated(WindowEvent we)  
void windowDeiconified(WindowEvent we)  
void windowIconified(WindowEvent we)  
void windowOpened(WindowEvent we)
```

## Adapter Classes

Java provides a special feature, called an adapter class, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the `MouseMotionAdapter` class has two methods, `mouseDragged()` and `mouseMoved()`, which are the methods defined by the `MouseMotionListener` interface. If you were interested in only mouse drag events, then you could simply extend `MouseMotionAdapter` and override `mouseDragged()`. The empty implementation of `mouseMoved()` would handle the mouse motion events for the user.

## JAVA TUTORIAL

### Closing AWT Window

We can close the AWT Window or Frame by calling *dispose()* or *System.exit()* inside *windowClosing()* method. The *windowClosing()* method is found in **WindowListener** interface and **WindowAdapter** class.

The **WindowAdapter** class implements **WindowListener** interfaces. It provides the default implementation of all the 7 methods of **WindowListener** interface. To override the *windowClosing()* method, you can either use **WindowAdapter** class or **WindowListener** interface.

If you implement the **WindowListener** interface, you will be forced to override all the 7 methods of **WindowListener** interface. So, it is better to use **WindowAdapter** class.

Different ways to override *windowClosing()* method

There are many ways to override *windowClosing()* method:

- By anonymous class
- By inheriting **WindowAdapter** class
- By implementing **WindowListener** interface

### Button, Label, TextField, TextArea, CheckBox, CheckBoxGroup

**Button**

## JAVA TUTORIAL

Button is a control component that has a label and generates an event when pressed. When a button is pressed and released, AWT sends an instance of ActionEvent to the button, by calling processEvent on the button. The button's processEvent method receives all events for the button; it passes an action event along by calling its own processActionEvent method. The latter method passes the action event on to any action listeners that have registered an interest in action events generated by this button.

If an application wants to perform some action based on a button being pressed and released, it should implement ActionListener and register the new listener to receive events from this button, by calling the button's addActionListener method. The application can make use of the button's action command as a messaging protocol.

### Class declaration

Following is the declaration for **java.awt.Button** class:

```
public class Button extends Component implements AccessibleClass
```

### **Constructors**

1. **1Button()**
2. Constructs a button with an empty string for its label.
3. **2Button(String text)**

## JAVA TUTORIAL

Constructs a new button with specified label.

Class methods

1. **void addActionListener(ActionListener l)** - Adds the specified action listener to receive action events from this button.
2. **void addNotify()** - Creates the peer of the button.
3. **AccessibleContext getAccessibleContext()** - Gets the AccessibleContext associated with this Button.
4. **String getActionCommand()** - Returns the command name of the action event fired by this button.
5. **ActionListener[] getActionListeners()** - Returns an array of all the action listeners registered on this button.
6. **String getLabel()** - Gets the label of this button
7. **<T extends EventListener> T[] getListeners(Class<T> listenerType)** - Returns an array of all the objects currently registered as FooListeners upon this Button.
8. **protected String paramString()** - Returns a string representing the state of this Button.
9. **protected void process.ActionEvent(ActionEvent e)**  
- Processes action events occurring on this button by dispatching them to any registered ActionListener objects.
10. **protected void processEvent(AWTEvent e)** - Processes events on this button.

**JAVA TUTORIAL**

11. **void removeActionListener(ActionListener l)** -  
Removes the specified action listener so that it no longer receives action events from this button.
12. **void setActionCommand(String command)** - Sets the command name for the action event fired by this button.
13. **void setLabel(String label)** - Sets the button's label to be the specified string.

***Methods inherited***

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

***Label***

Label is a passive control because it does not create any event when accessed by the user. The label control is an object of Label. A label displays a single line of read-only text. However, the text can be changed by the application programmer but cannot be changed by the end user in any way.

**Class declaration**

Following is the declaration for **java.awt.Label** class:

```
public class Label extends Component implements AccessibleField
```

## JAVA TUTORIAL

Following are the fields for **java.awt.Component** class:

- **static int CENTER** -- Indicates that the label should be centered.
- **static int LEFT** -- Indicates that the label should be left justified.
- **static int RIGHT** -- Indicates that the label should be right justified.

Class constructors

1 **Label()** - Constructs an empty label.

2 **Label(String text)** - Constructs a new label with the specified string of text, left justified.

3 **Label(String text, int alignment)** - Constructs a new label that presents the specified string of text with the specified alignment.

Class methods

1 **void addNotify()** - Creates the peer for this label.

2 **AccessibleContext getAccessibleContext()** - Gets the AccessibleContext associated with this Label.

3 **int getAlignment()** - Gets the current alignment of this label.

4 **String getText()** - Gets the text of this label.

## JAVA TUTORIAL

5 **protected String paramString()** - Returns a string representing the state of this Label.

6 **void setAlignment(int alignment)** - Sets the alignment for this label to the specified alignment.

7 **void setText(String text)** - Sets the text for this label to the specified text.

### Methods inherited

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

### *Textfield*

The textField component allows the user to edit single line of text. When the user types a key in the text field the event is sent to the TextField. The key event may be key pressed, Key released or key typed. The key event is passed to the registered KeyListener. It is also possible to for an ActionEvent if the ActionEvent is enabled on the textfield then ActionEvent may be fired by pressing the return key.

### Class declaration

Following is the declaration for **java.awt.TextField** class:

```
public class TextField extends TextComponentClass
```

## JAVA TUTORIAL

### Constructors

1. **TextField()** - Constructs a new text field.
2. **TextField(int columns)** - Constructs a new empty text field with the specified number of columns.
3. **TextField(String text)** - Constructs a new text field initialized with the specified text.
4. **TextField(String text, int columns)** - Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

### Class methods

1. **void addActionListener(ActionListener l)** - Adds the specified action listener to receive action events from this text field.
2. **void addNotify()** - Creates the TextField's peer.
3. **boolean echoCharIsSet()** - Indicates whether or not this text field has a character set for echoing.
4. **AccessibleContext getAccessibleContext()** - Gets the AccessibleContext associated with this TextField.
5. **ActionListener[] getActionListeners()** - Returns an array of all the action listeners registered on this textfield.
6. **int getColumns()** - Gets the number of columns in this text field.

**JAVA TUTORIAL**

7. **char getEchoChar()** - Gets the character that is to be used for echoing.
8. **<T extends EventListener> T[] getListeners(Class<T> listenerType)** - Returns an array of all the objects currently registered as FooListeners upon this TextField.
9. **Dimension getMinimumSize()** - Gets the minimum dimensions for this text field.
10. **Dimension getMinimumSize(int columns)** Gets the minimum dimensions for a text field with the specified number of columns.
11. **Dimension getPreferredSize()** - Gets the preferred size of this text field.
12. **Dimension getPreferredSize(int columns)** - Gets the preferred size of this text field with the specified number of columns.
13. **Dimension minimumSize()** - Deprecated. As of JDK version 1.1, replaced by getMinimumSize().
14. **Dimension minimumSize(int columns)** - Deprecated. As of JDK version 1.1, replaced by getMinimumSize(int).
15. **protected String paramString()** - Returns a string representing the state of this TextField.
16. **Dimension preferredSize()** - Deprecated. As of JDK version 1.1, replaced by getPreferredSize().
17. **Dimension preferredSize(int columns)** - Deprecated. As of JDK version 1.1, replaced by getPreferredSize(int).

## JAVA TUTORIAL

18. **protected void processActionEvent(ActionEvent e)**  
- Processes action events occurring on this text field by dispatching them to any registered ActionListener objects.
19. **protected void processEvent(AWTEvent e)** -  
Processes events on this text field.
20. **void removeActionListener(ActionListener l)** -  
Removes the specified action listener so that it no longer receives action events from this text field.
21. **void setColumns(int columns)** - Sets the number of columns in this text field.
22. **void setEchoChar(char c)** - Sets the echo character for this text field.
23. **void setEchoCharacter(char c)** - Deprecated. As of JDK version 1.1, replaced by setEchoChar(char).
24. **void setText(String t)** - Sets the text that is presented by this text component to be the specified text.

### Methods inherited

This class inherits methods from the following classes:

- java.awt.TextComponent

## JAVA TUTORIAL

- `java.awt.Component`
- `java.lang.Object`

### ***Checkbox***

Checkbox control is used to turn an option on(true) or off(false). There is label for each checkbox representing what the checkbox does. The state of a checkbox can be changed by clicking on it.

Class declaration

Following is the declaration for **`java.awt.Checkbox`** class:

```
public class Checkbox extends Component implements  
ItemSelectable, AccessibleClass
```

### ***Constructors***

1. **`Checkbox()`** - Creates a check box with an empty string for its label.
2. **`Checkbox(String label)`** - Creates a check box with the specified label.
3. **`Checkbox(String label, boolean state)`** - Creates a check box with the specified label and sets the specified state.
4. **`Checkbox(String label, boolean state, CheckboxGroup group)`** - Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group.

## JAVA TUTORIAL

5. **Checkbox(String label, CheckboxGroup group, boolean state)** - Creates a check box with the specified label, in the specified check box group, and set to the specified state.

### *Class methods*

1. **void addItemListener(ItemListener l)** - Adds the specified item listener to receive item events from this check box.
2. **void addNotify()** - Creates the peer of the Checkbox.
3. **AccessibleContext getAccessibleContext()** - Gets the AccessibleContext associated with this Checkbox.
4. **CheckboxGroup getCheckboxGroup()** - Determines this check box's group.
5. **ItemListener[] getItemListeners()** - Returns an array of all the item listeners registered on this checkbox.
6. **String getLabel()** - Gets the label of this check box.
7. **<T extends EventListener>T[] getListeners(Class<T> listenerType)** - Returns an array of all the objects currently registered as FooListeners upon this Checkbox.
8. **Object[] getSelectedObjects()** - Returns an array (length 1) containing the checkbox label or null if the checkbox is not selected.
9. **boolean getState()** - Determines whether this check box is in the **on** or **off** state.
10. **protected String paramString()** - Returns a string representing the state of this Checkbox.

**JAVA TUTORIAL**

11. **protected void processEvent(AWTEvent e)** - Processes events on this check box.
12. **protected void processItemEvent(ItemEvent e)** - Processes item events occurring on this check box by dispatching them to any registered ItemListener objects.
13. **void removeItemListener(ItemListener l)** - Removes the specified item listener so that the item listener no longer receives item events from this check box.
14. **void setCheckboxGroup(CheckboxGroup g)** - Sets this check box's group to the specified check box group.
15. **void setLabel(String label)** - Sets this check box's label to be the string argument.
16. **void setState(boolean state)** - Sets the state of this check box to the specified state.

***Methods inherited***

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

***Checkbox group***

The CheckboxGroup class is used to group the set of checkbox.

Class declaration

## JAVA TUTORIAL

Following is the declaration for **java.awt.CheckboxGroup** class:

```
public class CheckboxGroup extends Object  
implements Serializable
```

Class comstructors

1. **CheckboxGroup()** - Creates a new instance of CheckboxGroup.

### *Class methods*

2. **Checkbox getCurrent()** - Deprecated. As of JDK version 1.1, replaced by getSelectedCheckbox().
3. **Checkbox getSelectedCheckbox()** - Gets the current choice from this check box group.
4. **void setCurrent(Checkbox box)** - Deprecated. As of JDK version 1.1, replaced by setSelectedCheckbox(Checkbox).
5. **void setSelectedCheckbox(Checkbox box)** - Sets the currently selected check box in this group to be the specified check box.
6. **String toString()** - Returns a string representation of this check box group, including the value of its current selection.

### *Methods inherited*

This class inherits methods from the following classes:

- java.lang.Object

**JAVA TUTORIAL**

## Choice, List, Canvas, Scrollbar, Menu, MenuItem

### ***Choice***

Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.

### ***Class declaration***

Following is the declaration for **java.awt.Choice** class:

```
public class Choice extends Component implements  
ItemSelectable, AccessibleClass constructors
```

### ***Constructor***

1. **Choice()** ()  
Creates a new choice menu.

### **Class methods**

1. **void add(String item)**  
Adds an item to this Choice menu.
2. **void addItem(String item)**  
Obsolete as of Java 2 platform v1.1.
3. **void addItemClickListener(ItemListener l)**  
Adds the specified item listener to receive item events from this Choice menu.
4. **void addNotify()**  
Creates the Choice's peer.

**JAVA TUTORIAL****5. int countItems()**

Deprecated. As of JDK version 1.1, replaced by getItemCount().

**6. AccessibleContext getAccessibleContext()**

Gets the AccessibleContext associated with this Choice.

**7. String getItem(int index)**

Gets the string at the specified index in this Choice menu.

**8. int getItemCount()**

Returns the number of items in this Choice menu.

**9. ItemListener[] getItemListeners()**

Returns an array of all the item listeners registered on this choice.

**10. <T extends EventListener> T[] getListeners(Class<T> listenerType)**

Returns an array of all the objects currently registered as FooListeners upon this Choice.

**11. int getSelectedIndex()**

Returns the index of the currently selected item.

**12. String getSelectedItem()**

Gets a representation of the current choice as a string.

**13. Object[] getSelectedObjects()**

Returns an array (length 1) containing the currently selected item.

**14. void insert(String item, int index)**

Inserts the item into this choice at the specified position.

**JAVA TUTORIAL****15. `protected String paramString()`**

Returns a string representing the state of this Choice menu.

**16. `protected void processEvent(AWTEvent e)`**

Processes events on this choice.

**17. `protected void processItemEvent(ItemEvent e)`**

Processes item events occurring on this Choice menu by dispatching them to any registered ItemListener objects.

**18. `void remove(int position)`**

Removes an item from the choice menu at the specified position.

**19. `void remove(String item)`**

Removes the first occurrence of item from the Choice menu.

**20. `void removeAll()`**

Removes all items from the choice menu.

**21. `void removeItemListener(ItemListener l)`**

Removes the specified item listener so that it no longer receives item events from this Choice menu.

**22. `void select(int pos)`**

Sets the selected item in this Choice menu to be the item at the specified position.

**23. `void select(String str)`**

Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.

***Methods inherited***

## JAVA TUTORIAL

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

### *List*

The List represents a list of text items. The list can be configured that user can choose either one item or multiple items.

### *Class declaration*

Following is the declaration for **java.awt.List** class:

```
public class List extends Component implements  
ItemSelectable, AccessibleClass constructors
```

### *Constructors:*

1. **List()**  
Creates a new scrolling list.
2. **List(int rows)**  
Creates a new scrolling list initialized with the specified number of visible lines.
3. **List(int rows, boolean multipleMode)**  
Creates a new scrolling list initialized to display the specified number of rows.

**JAVA TUTORIAL*****Class methods***

<T extends EventListener> T[] getListeners(Class<T> listenerType)

Returns an array of all the objects currently registered as FooListeners upon this List.

**Method & Description****1. void add(String item)**

Adds the specified item to the end of scrolling list.

**2. void add(String item, int index)**

Adds the specified item to the the scrolling list at the position indicated by the index.

**3. void addActionListener(ActionListener l)**

Adds the specified action listener to receive action events from this list.

**4. void addItem(String item)**

Deprecated. replaced by add(String).

**5. void addItem(String item, int index)**

Deprecated. replaced by add(String, int).

**6. void addItemSelectedListener(ItemListener l)**

Adds the specified item listener to receive item events from this list.

**7. void addNotify()**

Creates the peer for the list.

**8. boolean allowsMultipleSelections()**

Deprecated. As of JDK version 1.1, replaced by isMultipleMode().

**JAVA TUTORIAL****9. void clear()**

Deprecated. As of JDK version 1.1, replaced by removeAll().

**10. int countItems()**

Deprecated. As of JDK version 1.1, replaced by getItemCount().

**11. void delItem(int position)**

Deprecated. replaced by remove(String) and remove(int).

**12. void delItems(int start, int end)**

Deprecated. As of JDK version 1.1, Not for public use in the future. This method is expected to be retained only as a package private method.

**13. void deselect(int index)**

Deselects the item at the specified index.

**14. AccessibleContext getAccessibleContext()**

Gets the AccessibleContext associated with this List.

**15. ActionListener[] getActionListeners()**

Returns an array of all the action listeners registered on this list.

**16. String getItem(int index)**

Gets the item associated with the specified index.

**17. int getItemCount()**

Gets the number of items in the list.

**18. ItemListener[] getItemListeners()**

Returns an array of all the item listeners registered on this list.

**19. String[] getItems()**

**JAVA TUTORIAL**

- Gets the items in the list.
20. **Dimension getMinimumSize()**  
Determines the minimum size of this scrolling list.
21. **Dimension getMinimumSize(int rows)**  
Gets the minumum dimensions for a list with the specified number of rows.
22. **Dimension getPreferredSize()**  
Gets the preferred size of this scrolling list.
23. **Dimension getPreferredSize(int rows)**  
Gets the preferred dimensions for a list with the specified number of rows.
24. **int getRows()**  
Gets the number of visible lines in this list.
25. **int getSelectedIndex()**  
Gets the index of the selected item on the list,
26. **int[] getSelectedIndexes()**  
Gets the selected indexes on the list.
28. **String getSelectedItem()**  
Gets the selected item on this scrolling list.
29. **String[] getSelectedItems()**  
Gets the selected items on this scrolling list.
30. **Object[] getSelectedObjects()**  
Gets the selected items on this scrolling list in an array of Objects.
31. **int getVisibleIndex()**  
Gets the index of the item that was last made visible by the method makeVisible.
32. **boolean isIndexSelected(int index)**

**JAVA TUTORIAL**

Determines if the specified item in this scrolling list is selected.

**33. boolean isMultipleMode()**

Determines whether this list allows multiple selections.

**34. boolean isSelected(int index)**

Deprecated. As of JDK version 1.1, replaced by isIndexSelected(int).

**35. void makeVisible(int index)**

Makes the item at the specified index visible.

**36. Dimension minimumSize()**

Deprecated. As of JDK version 1.1, replaced by getMinimumSize().

**37. Dimension minimumSize(int rows)**

Deprecated. As of JDK version 1.1, replaced by getMinimumSize(int).

**38. protected String paramString()**

Returns the parameter string representing the state of this scrolling list.

**39. Dimension preferredSize()**

Deprecated. As of JDK version 1.1, replaced by getPreferredSize().

**40. Dimension preferredSize(int rows)**

Deprecated. As of JDK version 1.1, replaced by getPreferredSize(int).

**41. protected void processActionEvent(ActionEvent e)**

Processes action events occurring on this component by dispatching them to any registered ActionListener objects.

## JAVA TUTORIAL

**42. `protected void processEvent(AWTEvent e)`**

Processes events on this scrolling list.

**43. `protected void processItemEvent(ItemEvent e)`**

Processes item events occurring on this list by dispatching them to any registered ItemListener objects.

**44. `void remove(int position)`**

Removes the item at the specified position from this scrolling list.

**45. `void remove(String item)`**

Removes the first occurrence of an item from the list.

**46. `void removeActionListener(ActionListener l)`**

Removes the specified action listener so that it no longer receives action events from this list.

**47. `void removeAll()`**

Removes all items from this list.

**48. `void removeItemListener(ItemListener l)`**

Removes the specified item listener so that it no longer receives item events from this list.

**49. `void removeNotify()`**

Removes the peer for this list.

**50. `void replaceItem(String newValue, int index)`**

Replaces the item at the specified index in the scrolling list with the new string.

**51. `void select(int index)`**

Selects the item at the specified index in the scrolling list.

**52. `void setMultipleMode(boolean b)`**

## JAVA TUTORIAL

Sets the flag that determines whether this list allows multiple selections.

### 53. **void setMultipleSelections(boolean b)**

Deprecated. As of JDK version 1.1, replaced by setMultipleMode(boolean).

### *Methods inherited*

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

### *Canvas*

Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.

Class declaration

Following is the declaration for **java.awt.Canvas** class:

```
public class Canvas extends Component implements  
AccessibleClass constructors
```

### *Constructors*

#### 1. **Canvas()**

Constructs a new Canvas.

#### 2. **Canvas(GraphicsConfiguration config)**

Constructs a new Canvas given a GraphicsConfiguration object.

**JAVA TUTORIAL***Class methods*

1. **void addNotify()**  
Creates the peer of the canvas.
2. **void createBufferStrategy(int numBuffers)**  
Creates a new strategy for multi-buffering on this component.
3. **void createBufferStrategy(int numBuffers, BufferCapabilities caps)**  
Creates a new strategy for multi-buffering on this component with the required buffer capabilities.
4. **AccessibleContext getAccessibleContext()**  
Gets the AccessibleContext associated with this Canvas.
5. **BufferStrategy getBufferStrategy()**  
Returns the BufferStrategy used by this component.
6. **void paint(Graphics g)**  
Paints this canvas.
7. **void update(Graphics g)**  
Updates this canvas.

*Methods inherited*

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

*Scrollbar*

## JAVA TUTORIAL

Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

Class declaration

Following is the declaration for **java.awt.Scrollbar** class:

```
public class Scrollbar extends Component implements  
Adjustable, AccessibleField
```

Following are the fields for **java.awt.Image** class:

- **static int HORIZONTAL** --A constant that indicates a horizontal scroll bar.
- **static int VERTICAL** --A constant that indicates a vertical scroll bar.

Class constructors

1. **Scrollbar()**

Constructs a new vertical scroll bar.

2. **Scrollbar(int orientation)**

Constructs a new scroll bar with the specified orientation.

3. **Scrollbar(int orientation, int value, int visible, int minimum, int maximum)**

Constructs a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum values.

*Class methods*

**JAVA TUTORIAL**

1. **void addAdjustmentListener(AdjustmentListener l)**  
Adds the specified adjustment listener to receive instances of AdjustmentEvent from this scroll bar.
2. **void addNotify()**  
Creates the Scrollbar's peer.
3. **int getBlockIncrement()**  
Gets the block increment of this scroll bar.
4. **int getLineIncrement()**  
Deprecated. As of JDK version 1.1, replaced by getUnitIncrement().
5. **int getMaximum()**  
Gets the maximum value of this scroll bar.
6. **int getMinimum()**  
Gets the minimum value of this scroll bar.
7. **int getOrientation()**  
Returns the orientation of this scroll bar.
8. **int getPageIncrement()**  
Deprecated. As of JDK version 1.1, replaced by getBlockIncrement().
9. **int getUnitIncrement()**  
Gets the unit increment for this scrollbar.
10. **int getValue()**  
Gets the current value of this scroll bar.
11. **boolean getValueIsAdjusting()**  
Returns true if the value is in the process of changing as a result of actions being taken by the user.

**JAVA TUTORIAL****12. int getVisible()**

Deprecated. As of JDK version 1.1, replaced by getVisibleAmount().

**13. int getVisibleAmount()**

Gets the visible amount of this scroll bar.

**14. protected String paramString()**

Returns a string representing the state of this Scrollbar.

**15. protected void processAdjustmentEvent(AdjustmentEvent e)**

Processes adjustment events occurring on this scrollbar by dispatching them to any registered AdjustmentListener objects.

**16. protected void processEvent(AWTEvent e)**

Processes events on this scroll bar.

**17. void****removeAdjustmentListener(AdjustmentListener l)**

Removes the specified adjustment listener so that it no longer receives instances of AdjustmentEvent from this scroll bar.

**18. void setBlockIncrement(int v)**

Sets the block increment for this scroll bar.

**19. void setLineIncrement(int v)**

Deprecated. As of JDK version 1.1, replaced by setUnitIncrement(int).

**20. void setMaximum(int newMaximum)**

Sets the maximum value of this scroll bar.

**21. void setMinimum(int newMinimum)**

Sets the minimum value of this scroll bar.

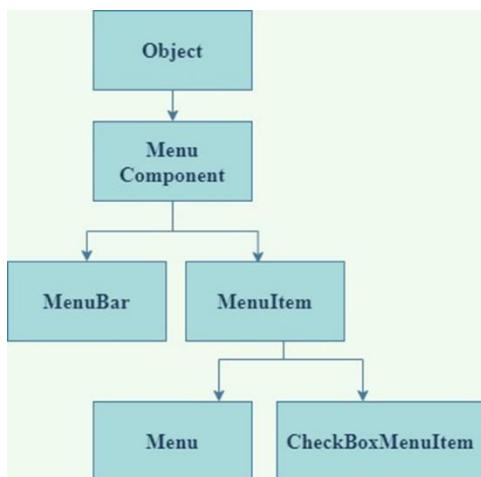
## JAVA TUTORIAL

22. **void setOrientation(int orientation)**  
Sets the orientation for this scroll bar.
23. **void setPageIncrement(int v)**  
Deprecated. As of JDK version 1.1, replaced by setBlockIncrement().
24. **void setUnitIncrement(int v)**  
Sets the unit increment for this scroll bar.
25. **void setValue(int newValue)**  
Sets the value of this scroll bar to the specified value.
26. **void setValueIsAdjusting(boolean b)**  
Sets the valueIsAdjusting property.
27. **void setValues(int value, int visible, int minimum, int maximum)**  
Sets the values of four properties for this scroll bar: value, visibleAmount, minimum, and maximum.
28. **void setVisibleAmount(int newAmount)**  
Sets the visible amount of this scroll bar.
29. **AccessibleContext getAccessibleContext()**  
Gets the AccessibleContext associated with this Scrollbar.
30. **AdjustmentListener[] getAdjustmentListeners()**  
Returns an array of all the adjustment listeners registered on this scrollbar.
31. **<T extends EventListener>T[] getListeners(Class<T> listenerType)**  
Returns an array of all the objects currently registered as FooListeners upon this Scrollbar.

**JAVA TUTORIAL*****Methods inherited***

This class inherits methods from the following classes:

- java.awt.Component
- java.lang.Object

***Menu***

As we know that every top-level window has a menu bar associated with it. This menu bar consists of various menu choices available to the end user. Further each choice contains list of options which is called drop down menus. **Menu** and **MenuItem** controls are subclass of **MenuComponent** class.

## JAVA TUTORIAL

### ***Menu Controls***

1. **MenuComponent**  
It is the top level class for all menu related controls.
2. **MenuBar**  
The **MenuBar** object is associated with the top-level window.
3. **MenuItem**  
The items in the menu must belong to the **MenuItem** or any of its subclass.
4. **Menu**  
The **Menu** object is a pull-down menu component which is displayed from the menu bar.
5. **CheckboxMenuItem**  
**CheckboxMenuItem** is subclass of **MenuItem**.
6. **PopupMenu**  
**PopupMenu** can be dynamically popped up at a specified position within a component.

### ***Menubar***

The **MenuBar** class represents the actual item in a menu. All items in a menu should derive from class **MenuItem**, or one of its subclasses. By default, it embodies a simple labeled menu item.

#### Class declaration

Following is the declaration for **java.awt.MenuItem** class:

## JAVA TUTORIAL

```
public class MenuItem extends MenuComponent  
implements AccessibleClass constructors
```

Constructors

1. **MenuItem()**

Constructs a new MenuItem with an empty label and no keyboard shortcut.

2. **MenuItem(String label)**

Constructs a new MenuItem with the specified label and no keyboard shortcut.

3. **MenuItem(String label, MenuShortcut s)**

Create a menu item with an associated keyboard shortcut.

Class methods

1. **void addActionListener(ActionListener l)**

Adds the specified action listener to receive action events from this menu item.

2. **void addNotify()**

Creates the menu item's peer.

3. **void deleteShortcut()**

Delete any MenuShortcut object associated with this menu item.

4. **void disable()**

Deprecated. As of JDK version 1.1, replaced by setEnabled(boolean).

5. **protected void disableEvents(long eventsToDisable)**

**JAVA TUTORIAL**

Disables event delivery to this menu item for events defined by the specified event mask parameter.

**6. void enable()**

Deprecated. As of JDK version 1.1, replaced by setEnabled(boolean).

**7. void enable(boolean b)**

Deprecated. As of JDK version 1.1, replaced by setEnabled(boolean).

**8. protected void enableEvents(long eventsToEnable)**

Enables event delivery to this menu item for events to be defined by the specified event mask parameter.

**9. AccessibleContext getAccessibleContext()**

Gets the AccessibleContext associated with this MenuItem.

**10. String getActionCommand()**

Gets the command name of the action event that is fired by this menu item.

**11. ActionListener[] getActionListeners()**

Returns an array of all the action listeners registered on this menu item.

**12. String getLabel()**

Gets the label for this menu item.

**13. EventListener[] getListeners(Class listenerType)**

Returns an array of all the objects currently registered as FooListeners upon this MenuItem.

**14. MenuShortcut getShortcut()**

Get the MenuShortcut object associated with this menu item.

## JAVA TUTORIAL

### 15. **boolean isEnabled()**

Checks whether this menu item is enabled.

### 16. **String paramString()**

Returns a string representing the state of this MenuItem.

### 17. **protected void processActionEvent(ActionEvent e)**

Processes action events occurring on this menu item, by dispatching them to any registered ActionListener objects.

### 18. **protected void processEvent(AWTEvent e)**

Processes events on this menu item.

### 19. **void removeActionListener(ActionListener l)**

Removes the specified action listener so it no longer receives action events from this menu item.

### 20. **void setActionCommand(String command)**

Sets the command name of the action event that is fired by this menu item.

### 21. **void setEnabled(boolean b)**

Sets whether or not this menu item can be chosen.

### 22. **void setLabel(String label)**

Sets the label for this menu item to the specified label.

### 23. **void setShortcut(MenuShortcut s)**

Set the MenuShortcut object associated with this menu item.

### *Methods inherited*

This class inherits methods from the following classes:

## JAVA TUTORIAL

- `java.awt.MenuComponent`
- `java.lang.Object`

**Popup menu, panel, dialog, toolkit, demo on event handling with AWT controls**

### ***Popup Menu***

Popup menu represents a menu which can be dynamically popped up at a specified position within a component.

### ***Class declaration***

Following is the declaration for **`java.awt.PopupMenu`** class:

```
public class CheckboxMenuItem extends MenuItem  
implements ItemSelectable, AccessibleClass constructors
```

### ***Constructor & Description***

#### **`PopupMenu()`**

Creates a new popup menu with an empty name.

#### **`PopupMenu(String label)`**

Creates a new popup menu with the specified name.

### ***Class methods***

1. S.N.Method & Description **`void addNotify()`**  
Creates the popup menu's peer.

## JAVA TUTORIAL

### 2. **AccessibleContext getAccessibleContext()**

Gets the AccessibleContext associated with this PopupMenu.

### 3. **MenuContainer getParent()**

Returns the parent container for this menu component.

### 4. **void show(Component origin, int x, int y)**

Shows the popup menu at the x, y position relative to an origin component.

### *Methods inherited*

This class inherits methods from the following classes:

- java.awt.MenuItem
- java.awt.MenuComponent
- java.lang.Object

### *Panel*

The class **Panel** is the simplest container class. It provides space in which an application can attach any other component, including other panels. It uses FlowLayout as default layout manager.

### *Class declaration*

Following is the declaration for **java.awt.Panel** class:

```
public class Panel extends Container implements  
AccessibleClass constructors
```

## JAVA TUTORIAL

S.N.Constructor & Description

### **Panel()**

Creates a new panel using the default layout manager.

### **Panel(LayoutManager layout)**

Creates a new panel with the specified layout manager.

### ***Class methods***

S.N.Method & Description

#### **void addNotify()**

Creates the Panel's peer.

#### **AccessibleContext getAccessibleContext()**

Gets the AccessibleContext associated with this Panel.

### ***Methods inherited***

This class inherits methods from the following classes:

- java.awt.Container
- java.awt.Component
- java.lang.Object

### ***Dialog***

Dialog control represents a top-level window with a title and a border used to take some form of input from the user.

## JAVA TUTORIAL

### Class declaration

Following is the declaration for **java.awt.Dialog** class:

```
public class Dialog extends WindowField
```

Following are the fields for **java.awt.Image** class:

**static Dialog.ModalityType DEFAULT\_MODALITY\_TYPE** –

Default modality type for modal dialogs.

### *Class constructors*

#### 1. **Dialog(Dialog owner)**

Constructs an initially invisible, modeless Dialog with the specified owner Dialog and an empty title.

#### 2. **Dialog(Dialog owner, String title)**

Constructs an initially invisible, modeless Dialog with the specified owner Dialog and title.

#### 3. **Dialog(Dialog owner, String title, boolean modal)**

Constructs an initially invisible Dialog with the specified owner Dialog, title, and modality.

#### 4. **Dialog(Dialog owner, String title, boolean modal, GraphicsConfiguration gc)**

Constructs an initially invisible Dialog with the specified owner Dialog, title, modality and GraphicsConfiguration.

#### 5. **Dialog(Frame owner)**

**JAVA TUTORIAL**

Constructs an initially invisible, modeless Dialog with the specified owner Frame and an empty title.

**6. Dialog(Frame owner, boolean modal)**

Constructs an initially invisible Dialog with the specified owner Frame and modality and an empty title.

**7. Dialog(Frame owner, String title)**

Constructs an initially invisible, modeless Dialog with the specified owner Frame and title.

**8. Dialog(Frame owner, String title, boolean modal)**

Constructs an initially invisible Dialog with the specified owner Frame, title and modality.

**9. Dialog(Frame owner, String title, boolean modal, GraphicsConfiguration gc)**

Constructs an initially invisible Dialog with the specified owner Frame, title, modality, and GraphicsConfiguration.

**10. Dialog(Window owner)**

Constructs an initially invisible, modeless Dialog with the specified owner Window and an empty title.

**11. Dialog(Window owner, Dialog.ModalityType modalityType)**

Constructs an initially invisible Dialog with the specified owner Window and modality and an empty title.

**12. Dialog(Window owner, String title)**

Constructs an initially invisible, modeless Dialog with the specified owner Window and title.

**JAVA TUTORIAL**

13. **Dialog(Window owner, String title, Dialog.ModalityType modalityType)**  
Constructs an initially invisible Dialog with the specified owner Window, title and modality.
14. **Dialog(Window owner, String title, Dialog.ModalityType modalityType, GraphicsConfiguration gc)**  
Constructs an initially invisible Dialog with the specified owner Window, title, modality and GraphicsConfiguration

***Class methods***

1. **void addNotify()**  
Makes this Dialog displayable by connecting it to a native screen resource.
2. **AccessibleContext getAccessibleContext()**  
Gets the AccessibleContext associated with this Dialog.
3. **Dialog.ModalityType getModalityType()**  
Returns the modality type of this dialog.
4. **String getTitle()**  
Gets the title of the dialog.
5. **void hide()**  
Deprecated. As of JDK version 1.5, replaced by setVisible(boolean).
6. **boolean isModal()**  
Indicates whether the dialog is modal.
7. **boolean isResizable()**  
Indicates whether this dialog is resizable by the user.
8. **boolean isUndecorated()**

**JAVA TUTORIAL**

- Indicates whether this dialog is undecorated.
- 9. **protected String paramString()**  
Returns a string representing the state of this dialog.
  - 10. **void setModal(boolean modal)**  
Specifies whether this dialog should be modal.
  - 11. **void setModalityType(Dialog.ModalityType type)**  
Sets the modality type for this dialog.
  - 12. **void setResizable(boolean resizable)**  
Sets whether this dialog is resizable by the user.
  - 13. **void setTitle(String title)**  
Sets the title of the Dialog.
  - 14. **void setUndecorated(boolean undecorated)**  
Disables or enables decorations for this dialog.
  - 15. **void setVisible(boolean b)**  
Shows or hides this Dialog depending on the value of parameter b.
  - 16. **void show()**  
Deprecated. As of JDK version 1.5, replaced by setVisible(boolean).
  - 17. **void toBack()**  
If this Window is visible, sends this Window to the back and may cause it to lose focus or activation if it is the focused or active Window.

***Methods inherited***

This class inherits methods from the following classes:

- java.awt.Window
- java.awt.Component
- java.lang.Object

## JAVA TUTORIAL

### *Toolkit*

Toolkit class is the abstract superclass of every implementation in the Abstract Window Toolkit. Subclasses of Toolkit are used to bind various components. It inherits Object class.

AWT Toolkit class declaration

1. **public abstract class Toolkit extends Object**

## Event handling with AWT

### Delegation event Model

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event.

## JAVA TUTORIAL

The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listeners that want to receive them.

### Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

### Points to remember about listener

- In order to design a listener class, we have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

### Callback Methods

## JAVA TUTORIAL

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represent an event method. In response to an event java JRE will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to its events the source must register itself to the listener.

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The `java.awt.event` package provides many event classes and Listener interfaces for event handling.

**JAVA TUTORIAL**

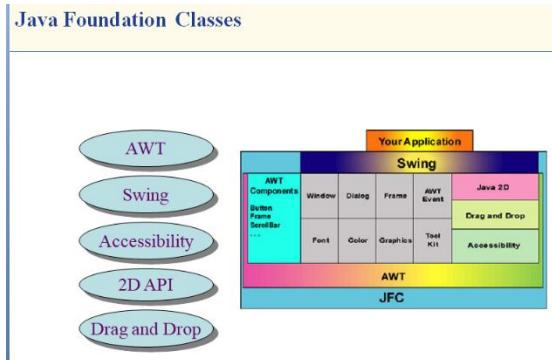
## **9 *Swing***

---

### **Java Swing**

Java is a relatively new programming language that uses an object-oriented approach that has become popular for developing Internet applications. However, Java had a weakness: it lacked sophisticated graphic user interface (GUI) components needed to make large-scale windowing applications. Java did have a GUI toolkit: The Abstract Window Toolkit (AWT), but it was not functional enough for full-scale graphic applications. The widget library was small and contained only the most basic of components. Those components only had basic functionality (such as buttons that could only have text labels not images). This meant that developers who wanted more had to spend considerable time creating classes to extend the functionality of the widgets provided by the AWT. In the fast-moving world of software development, this was a serious waste of resources. Also, the AWT components relied on native peers which meant that portability was hampered by minor differences in the way each operating system handled a particular widget, and the look-and-feel of the components was governed by the native operating system (OS).

## JAVA TUTORIAL



One of the weaknesses of the AWT is its use of heavyweight components. These are widgets which rely on native peers for their look-and-feel. These peer classes are linked to corresponding widgets and are responsible for talking to the native operating system. They say that operating system to create the widget, so it has the same appearance as a native one. This is why an AWT button on a Windows platform looks different from the same button displayed on a Mac. This reliance on the native OS proves troublesome for a number of reasons. Since the developer has no direct control over the look of components, changing the look of components is difficult. Also, Peer-driven widgets may behave differently depending on the platform. These differences are slight, but if they are a problem for the developer, they may require complex code to fix. This is one of the reasons that the AWT components have only basic functionality. More complex abilities would be too difficult to implement cross-platform. Since native peers are required, a common, cross-platform look & feel is impossible. This makes it awkward for developers who want their software to run on other platforms as documentation and testing have to be done for each OS they want to use. This would not be

## JAVA TUTORIAL

such a problem except that one of Java's strengths, that it is a cross-platform language.

Sun began a project to beef up Java in 1996. The result was the Java Foundation Classes released in 1998. Swing is the new and improved GUI toolkit included in the JFC. With Swing, Java can produce highly complex GUIs for industrial strength applications. Swing components are pure Java components no longer relying on their native peers and are therefore platform neutral. Project began late 1996. Active development since spring 1997. Beta in late 1997. Initial release March 1998 as part of the JFC.

## Difference between AWT & Swing

AWT	SWING
Platform dependent	Platform Independent
Heavy Weight	Lightweight
Does not support pluggable and feel	Supports pluggable look and feel
Less Components	More components
Don't follow MVC	Follows MVC

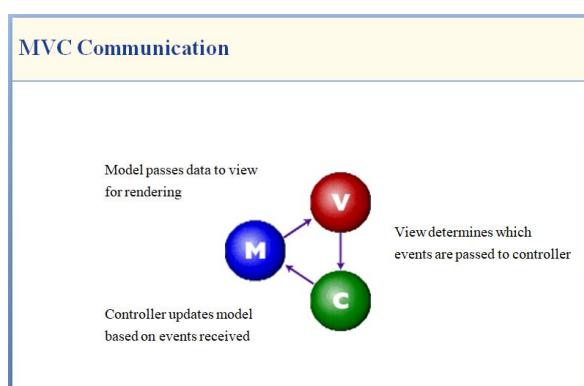
Swing -

- uses **lightweight** components
- uses a variant of the **Model View Controller Architecture (MVC)**
- has **Pluggable Look and Feel (PLAF)**

## JAVA TUTORIAL

- uses the **Delegation Event Model**
- components can have transparent portions
- components can be any shape, and can overlap each other
- **Look and Feel** drawn at runtime so can vary
- functionality is the same on all platforms
- though slower, preferred by the industry

The Java Foundation Classes are Sun's answer to Java's deficiencies. They were released in March 1998 and contained five main elements: Java2d, Accessibility, Drag and Drop, AWT, and Swing. Java2d has classes for more complex use of painting, shape, color, and fonts. Accessibility is a set of tools for implementing support for physically challenged users in an application that use non-traditional means for input and output. Drag and Drop is an API which allows GUI objects to be "dragged" via the mouse from one part of an application to another, between Java applications, or even between Java and other applications. The AWT is the original one from Java and is included to support code that already uses the AWT. In this session we primarily focus on Swing. Swing is designed for forms-based applications development. It provides a rich set of components and a framework to specify how to present GUIs that are visually independent of the platform.

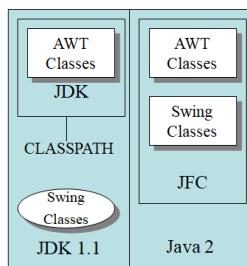
**JAVA TUTORIAL**

Swing has a number of new features to improve GUI design in applications. These are lightweight components, a variant of the Model-View-Controller Architecture (MVC), and Pluggable Look-and-Feel (PLAF). It also uses the Delegation Event Model from JDK 1.1. Swing fixes these problems by using only Java, no native code, to create components. These are called lightweight components since no peer classes are required. Instead, the components draw themselves on the screen. This self-reliance allows Swing components to do many things their AWT counterparts could not, such as using images on buttons. It also means that widgets not available through the native OS are possible. This is why Swing includes a host of new, highly functional widgets. These Swing components are distinguished by the J-prefix in front of the name (for example, a JButton is a Swing button class).

**JAVA TUTORIAL**

# Swing Controls

## How do I Use Swing?

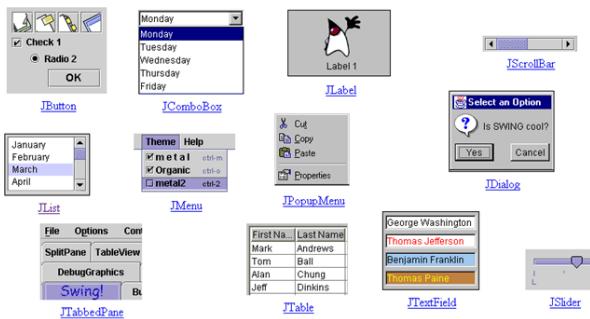


A Swing GUI consists of two key items: *components* and *containers*. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

## JAVA TUTORIAL

## Useful Components

- Click to add text



**JComponent** is the common root of most of the Swing GUI classes. It provides the guiding framework for GUI objects. It extends `java.awt.Container` class, and other objects (components) can be added to it. The **JComponent** class extends the `Container` class, which itself extends `Component`. The `Component` class includes everything from providing layout hints to supporting painting and events. The `Container` class has support for adding components to the container and laying them out.

**JComponent Features:** Few of the functionality provided by `JComponent` class to its descendants are:

Tool tips

## JAVA TUTORIAL

By specifying a string with the `setToolTipText( )` method, you can provide help to users of a component. When the cursor pauses over the component, the specified string is displayed in a small window that appears near the component.

### *Borders*

The `setBorder( )` method allows you to specify the border that a component displays around its edges.

### *Keyboard-generated actions*

Using the `registerKeyboardAction( )` method, you can enable the user to use the keyboard, instead of the mouse, to operate the GUI. The combination of character and modifier keys that the user must press to start an action is represented by a `KeyStroke` object. The resulting action event must be handled by an `ActionListener`.

**Application-wide Pluggable Look and Feel:** Behind the scenes, each `JComponent` object has a corresponding `ComponentUI` object that performs all the drawing, event handling, size determination, and so on for that `JComponent`. Exactly which `ComponentUI` object is used depends on the current look and feel, which you can set using the `UIManager.setLookAndFeel( )` method.

**Properties:** You can associate one or more properties (name/object pairs) with any `JComponent`. For example, a layout manager might use properties to associate a `constraints` object with each `JComponent` it manages. You put and get properties using the `putClientProperty( )` and `getClientProperty( )` methods.

## JAVA TUTORIAL

### Methods Inherited from Component Class

- get/setBackground()
- is/setEnabled()
- get/setFont()
- get/setForeground()
- get/setLayout()
- get/setLocationOnScreen()
- get/setName()
- get/setParent()
- is/setVisible()

### New Methods

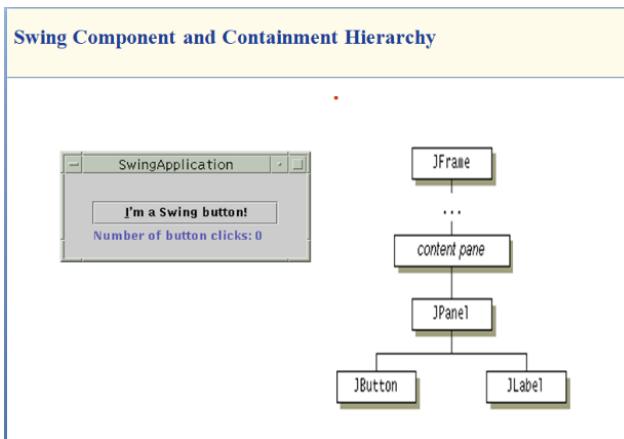
- get/setBounds()
- get/setSize()
- get/setLocation()
- get/setWidth()
- get/setHeight()
- get/setMaximumSize()
- get/setMinimumSize()
- get/setPreferredSize()

Methods to increase efficiency: JComponent has few methods that provide more efficient ways to get information than the JDK 1.1 API allowed. The methods include getX( ) and getY( ), which you can use instead of getLocation( ); and getWidth( ) and getHeight( ), which you can use instead of getSize( ). It also adds one argument forms of getBounds( ), getLocation( ), and getSize( ) for

**JAVA TUTORIAL**

which you specify the object to be modified and returned, letting you avoid unnecessary object creation. These methods have been added to Component for Java 2 (JDK 1.2).

## Java Swing Apps



Swing programs differ from both the console-based programs and the AWT-based programs. Swing programs also have special requirements that relate to threading. The best way to understand the structure of a Swing program is to work through an example. There are two types of Java programs in which Swing is typically used. The first is a desktop application. The second is the applet. In the process, it demonstrates several key features of Swing. It uses two Swing components: JFrame and JLabel. JFrame is the top-level container that is commonly used for Swing applications. JLabel is the Swing component that creates a label, which is a component that displays information. The label is Swing's

## JAVA TUTORIAL

simplest component because it is passive. That is, a label does not respond to user input. It just displays output. The program uses a JFrame container to hold an instance of a JLabel. The label displays a short text message.

Even the simplest Swing program like the one in slide will have multiple levels in its containment hierarchy. The root of the containment hierarchy is always a top-level container. The top-level container provides a place for its descendent Swing components to paint themselves. The frame is a top-level container. It exists mainly to provide a place for other Swing components to paint themselves. The other commonly used top-level containers are dialogs (JDialog) and applets (JApplet). The panel is an intermediate container. Its only purpose is to simplify the positioning of the button and label. Other intermediate Swing containers, such as scroll panes ( JScrollPane) and tabbed panes(JTabbedPane), typically play a more visible, interactive role in a program's GUI. The button and label are atomic components - components that exist not to hold random Swing components, but as self-sufficient entities that present bits of information to the user. Often, atomic components also get input from the user. The Swing API provides many atomic components, including combo boxes ( JComboBox), text fields ( JTextField), and tables ( JTable).

### Adding Components to Containers

```
frame = new JFrame(...);
button = new JButton(...);
label = new JLabel(...);
pane = new JPanel();
pane.add(button);
```

## JAVA TUTORIAL

```
pane.add(label);
frame.getContentPane().add(pane, BorderLayout.CENTER);
```

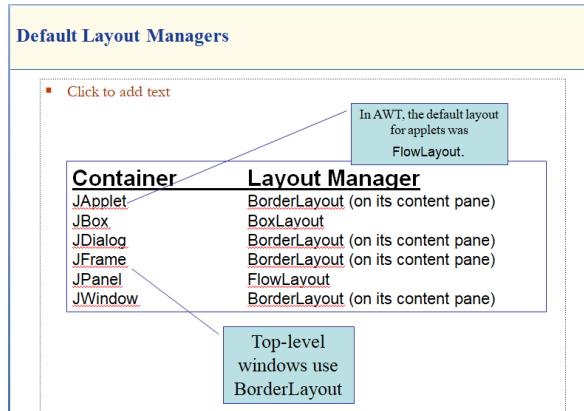
# Layout Manager

## Layout Managers

- Arrange widgets according to a pattern
- Can update containers to handle resizing of the container or internal widgets
- Make complex UIs possible

Layout management is the process of determining the size and position of components. By default, each container has a layout manager -- an object that performs layout management for the components within the container. Components can provide size and alignment hints to layout managers, but layout managers have the final say on the size and position of those components.

## JAVA TUTORIAL

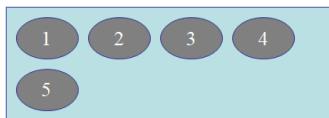
**Manager Description:**

- ***java.awt.BorderLayout*** - Arranges elements along the north, south, east, west, and in the center of the container.
- ***java.swing.BoxLayout*** - Arranges elements in a single row or single column.
- ***java.awt.CardLayout*** - Arranges elements like a stack of cards, with one visible at a time.
- ***java.awt.FlowLayout*** - Arranges elements left to right across the container.
- ***java.awt.GridBagLayout*** - Arranges elements in a grid of variable sized cells (complicated).
- ***java.awt.GridLayout*** - Arranges elements into a two-dimensional grid of equally sized cells.
- ***java.swing.OverlayLayout*** - Arranges elements on top of each other.

## JAVA TUTORIAL

### FlowLayout

- Arranges components from left to right and top to bottom
- Fits as many components in a row as possible before making a new row
- lets you specify alignment, horizontal and vertical spacing



FlowLayout puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, FlowLayout uses multiple rows. Within each row, components are centered (the default), left-aligned, or right-aligned as specified when the FlowLayout is created.

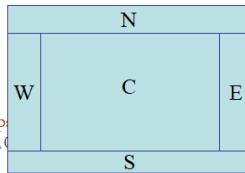
Example:

```
toolbar.setLayout(new  
FlowLayout(FlowLayout.LEFT));  
toolbar.add(playButton);  
toolbar.add(stopButton);
```

## JAVA TUTORIAL

### Border Layout

- Arranges components according to specified edges or the middle
  - NORTH
  - SOUTH
  - EAST
  - WEST
  - CENTER
- lets you specify horizontal and vertical spans
  - contentPanel.setLayout(new BorderLayout(0, 0));
  - contentPanel.add("Center", oPanel);
  - contentPanel.add("South", controlPanel);



A BorderLayout has five areas: north, south, east, west, and center. If you enlarge the window, the center area gets as much of the available space as possible. The areas expand only as much as necessary to fill all available space. Often, a container uses only one or two of the areas of the BorderLayout --just the center, or center and south.

For example,

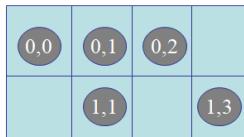
```
Container contentPane = getContentPane();  
//Use the content pane's default  
BorderLayout.  
//contentPane.setLayout (new  
BorderLayout()); //unnecessary  
contentPane.add(new JButton ("Button 1  
(NORTH)"), BorderLayout.NORTH);  
contentPane.add(new JButton ("CENTER")),  
BorderLayout.CENTER);
```

**JAVA TUTORIAL**

```
contentPane.add(new JButton("WEST"),  
BorderLayout.WEST);  
contentPane.add(new JButton("Long-Named  
Button 4 (SOUTH)"), BorderLayout.SOUTH);  
contentPane.add(new JButton("Button 5  
(EAST)"), BorderLayout.EAST);
```

**Grid Layout**

- Arranges components in a grid with specified rows and columns
- rows have same height and columns have same width



```
contentPanel.setLayout(new GridLayout(2, 4));  
contentPanel.add(starButton);  
contentPanel.add(stopButton);
```

A GridLayout places components in a grid of cells. Each component takes all the available space within its cell, and each cell is exactly the same size. If you resize the GridLayout window, you'll see that the GridLayout changes the cell size so that the cells are as large as possible, given the space available to the container.  
Example:

```
Container contentPane = getContentPane()
```

## JAVA TUTORIAL

```
contentPane.setLayout(new  
GridLayout(0,2));  
contentPane.add(new JButton("Button 1"));  
contentPane.add(new JButton("2"));  
contentPane.add(new JButton("Button 3"));  
ContentPane.add(new JButton("Long-Named  
Button 4"));  
contentPane.add(new JButton("Button 5"));
```

The constructor tells the GridLayout class to create an instance that has two columns and as many rows as necessary.

## Graphics in Swing

Swing was so successful that it has remained the primary Java GUI framework for over a decade. (And a decade is a long time in the fast-moving world of programming!) However, Swing was designed when the enterprise application dominated software development. Today, consumer applications, and especially mobile apps, have risen in importance, and such applications often demand a GUI that has “visual sparkle.” Furthermore, no matter the type of application, the trend is toward more exciting visual effects. To better handle these types of GUIs, a new approach was needed, and this lead to the creation of JavaFX. JavaFX is Java’s next-generation client platform and GUI framework. JavaFX provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs.

In general, the JavaFX framework has all of the good features of Swing. For example, JavaFX is lightweight. It can also support an MVC architecture. Much of what you already know about creating

## JAVA TUTORIAL

GUIs using Swing is conceptually applicable to JavaFX. That said, there are significant differences between the two. From a programmer's point of view, the first differences you notice between JavaFX and Swing are the organization of the framework and the relationship of the main components. Simply put, JavaFX offers a more streamlined, easier-to-use, updated approach. JavaFX also greatly simplifies the rendering of objects because it handles repainting automatically. It is no longer necessary for your program to handle this task manually. The preceding is not intended to imply that Swing is poorly designed. It is not. It is just that the art and science of programming has moved forward, and JavaFX has received the benefits of that evolution. Simply put, JavaFX facilitates a more visually dynamic approach to GUIs.

The JavaFX elements are contained in packages that begin with the **javafx** prefix. At the time of this writing, there are more than 30 JavaFX packages in its API library. Here are four examples:

- *javafx.application*,
- *javafx.stage*,
- *javafx.scene*, and
- *javafx.scene.layout*

Although we will only use a few of these packages in this chapter, you will want to spend some time browsing their capabilities. JavaFX offers a wide array of functionality.

The central metaphor implemented by JavaFX is the *stage*. As in the case of an actual stage play, a stage contains a *scene*. Thus, loosely speaking, a stage defines a space and a scene defines what goes in that space. Or, put another way, a stage is a container for

## JAVA TUTORIAL

scenes and a scene is a container for the items that comprise the scene. As a result, all JavaFX applications have at least one stage and one scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes. To create a JavaFX application, you will, at minimum, add at least one **Scene** object to a **Stage**. Let's look a bit more closely at these two classes. **Stage** is a top-level container. All JavaFX applications automatically have access to one **Stage**, called the *primary stage*. The primary stage is supplied by the run-time system when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.

The individual elements of a scene are called *nodes*. For example, a push button control is a node. However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a *parent node* or *branch node*. Nodes without children are terminal nodes and are called leaves. The collection of all nodes in a scene creates what is referred to as a *scene graph*, which comprises a *tree*. There is one special type of node in the scene graph, called the *root node*. This is the top-level node and is the only node in the scene graph that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node. The base class for all nodes is **Node**. There are several other classes that are, either directly or indirectly, subclasses of **Node**. These include **Parent**, **Group**, **Region**, and **Control**, to name a few.

## JAVA TUTORIAL

### *Layouts*

JavaFX provides several layout panes that manage the process of placing elements in a scene. For example, the **FlowPane** class provides a flow layout and the **GridPane** class supports a row/column grid-based layout. Several other layouts, such as **BorderPane** (which is similar to the AWT's **BorderLayout**), are available. The layout panes are packaged in **javafx.scene.layout**.

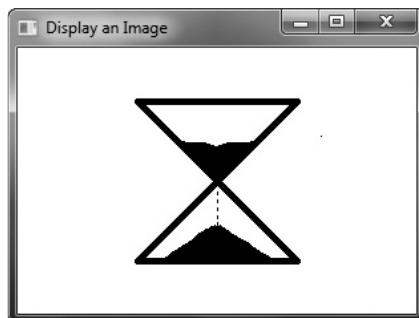
## Displaying Images

**Image** encapsulates the image, itself, and **ImageView** manages the display of an image. Both classes are packaged in **javafx.scene.image**.

The **Image** class loads an image from either an **InputStream**, a URL, or a path to the image file. **Image** defines several constructors; this is the one we will use:

`Image(String url)`

Here, *url* specifies a URL or a path to a file that supplies the image. The argument is assumed to refer to a path if it does not constitute a properly formed URL. Otherwise, the image is loaded from the URL. The examples that follow will load images from files on the local file system. Other constructors let you specify various options, such as the image's width and height. One other point: **Image** is not derived from **Node**. Thus, it cannot, itself, be part of a scene graph. Once you have an **Image**, you will use **ImageView** to display it. **ImageView** is derived from **Node**, which means that it can be part of a scene graph.

**JAVA TUTORIAL**

**ImageView** defines three constructors. The first one we will use is shown here:

*ImageView(Image image)*

This constructor creates an **ImageView** that uses *image* for its image.

For Example,

```
import java.awt.FlowLayout;
import java.awt.image.BufferedImage;
import java.net.URL;
import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Main {

    public static void main(String avg[]) throws Exception {
        BufferedImage img = ImageIO.read(new URL(
            "image.png"));
    }
}
```

**JAVA TUTORIAL**

```
ImageIcon icon = new ImageIcon(img);
JFrame frame = new JFrame();
frame.setLayout(new FlowLayout());
frame.setSize(500, 500);
JLabel lbl = new JLabel();
lbl.setIcon(icon);
frame.add(lbl);
frame.setVisible(true);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

**JAVA TUTORIAL**

## 10 *JDBC*

---

### Introduction

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as –

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

## JAVA TUTORIAL

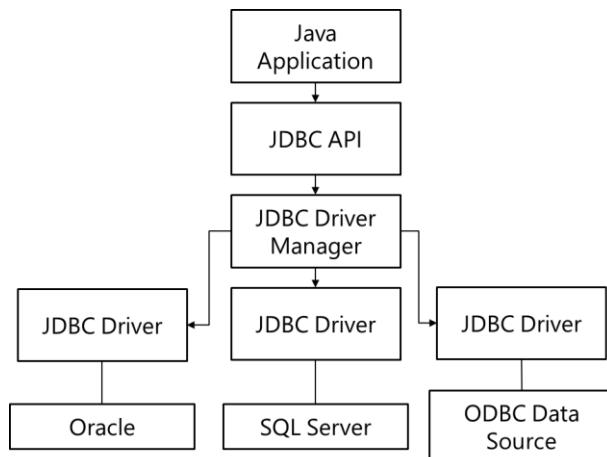
All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data. JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code. JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

1. JDBC-ODBC Bridge Driver,
2. Native Driver,
3. Network Protocol Driver, and
4. Thin Driver

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft. Before JDBC, ODBC API was the database API to connect and execute the query with the database. But ODBC API uses ODBC driver which is written in C language (i.e., platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

**JAVA TUTORIAL**

## Architecture & Querying with JDBC



### ***The JDBC Architecture:***

The JDBC API consists of two major sets of interfaces:

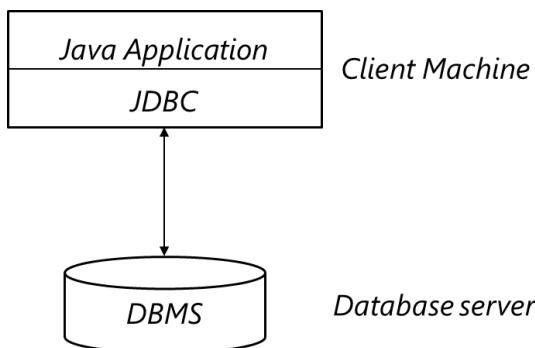
1. The first is the JDBC API for applications writers, and
2. The second is the lower-level JDBC driver API for driver writers.

As seen from the above diagram, the *DriverManager* class is the traditional management layer of JDBC, working between the user and the drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. In addition, the *DriverManager* class attends to

## JAVA TUTORIAL

things like driver login time limits and the printing of log and tracing messages.

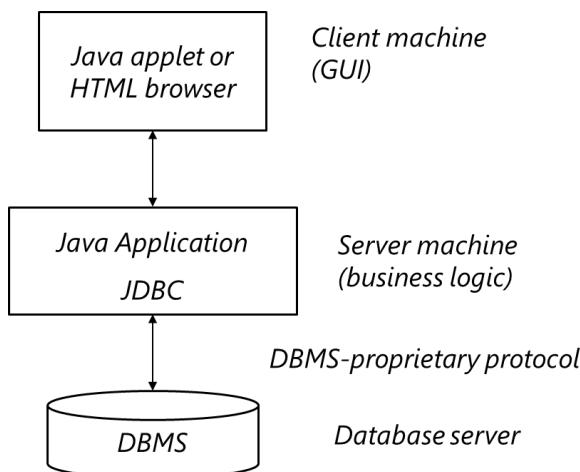
The JDBC API supports both *two-tier* and *three-tier* processing models for database access. In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.



In the *three-tier model*, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors

**JAVA TUTORIAL**

find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

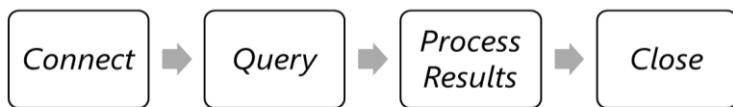


Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features. With enterprises increasingly using the Java programming language for writing

**JAVA TUTORIAL**

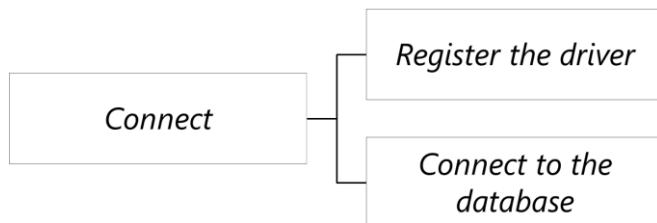
server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology is its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

## Querying with JDBC



In general, to process any SQL statement with JDBC, you follow these steps:

- Establishing a connection.
- Create a statement.
- Execute the query.
- Process the **ResultSet** object.
- Close the connection.

**JAVA TUTORIAL*****Stage 1: Connect***

The first thing you need to do is establish a connection with the DBMS (Database Management System) you want to use. This involves two steps:

1. loading the driver and
2. making the connection.

Now, we shall first see as to what is a **JDBC Driver**.

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps –

1. *Import JDBC Packages* – Add import statements to your Java program to import required classes in your Java code.
2. *Register JDBC Driver* – This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.

## JAVA TUTORIAL

3. *Database URL Formulation* – This is to create a properly formatted address that points to the database to which you wish to connect.
4. *Create Connection Object* – Finally, code a call to the DriverManager object's getConnection( ) method to establish actual database connection.

First, you need to establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver.

Typically, a JDBC application connects to a target data source using one of two classes:

### **DriverManager:**

This fully implemented class connects an application to a data source, which is specified by a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the class path. Note that your application must manually load any JDBC drivers prior to version 4.0.

### **DataSource:**

This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application. A DataSource object's properties are set so that it represents a particular data source.

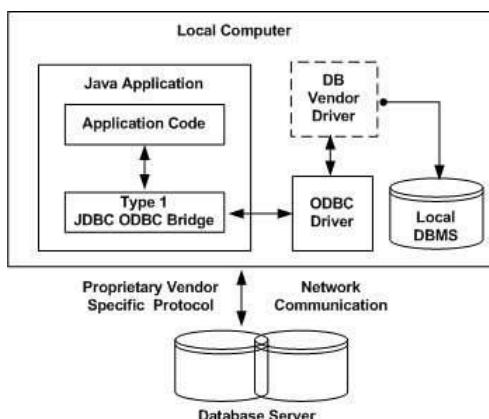
## JAVA TUTORIAL

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server. For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java. The `java.sql` package that ships with JDK, contains various classes with their behaviours defined and their actual implementations are done in third-party drivers. Third party vendors implement the `java.sql.Driver` interface in their database driver.

### JDBC Drivers Types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below –

#### Type 1 – JDBC-ODBC Bridge Driver



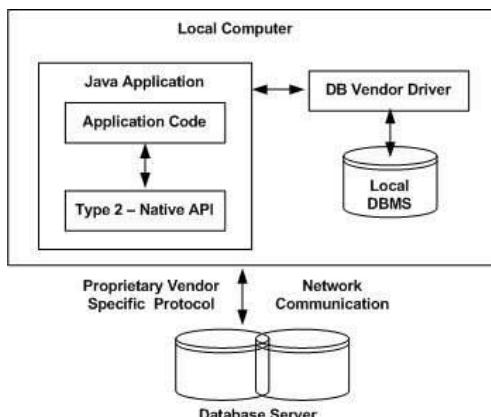
## JAVA TUTORIAL

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available. The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

### Type 2 – JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

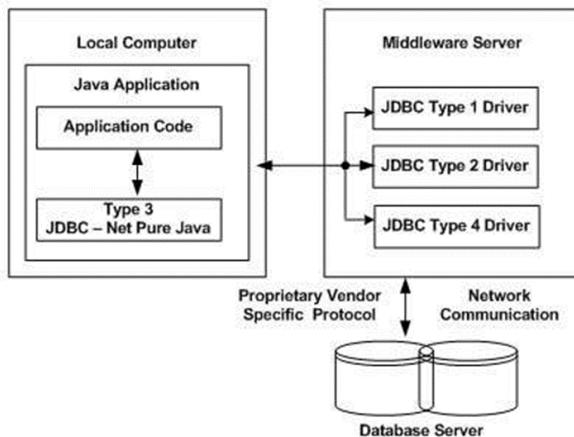


## JAVA TUTORIAL

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

### Type 3 – JDBC-Net pure Java

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.



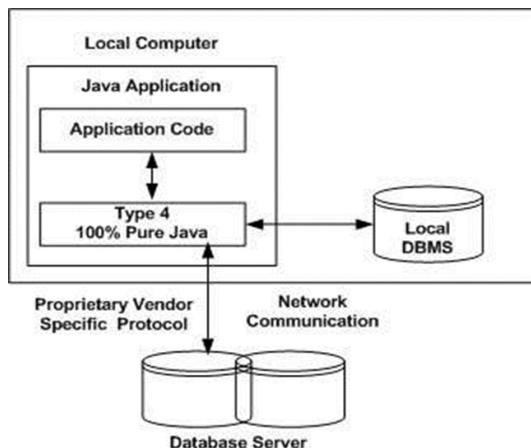
This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide

## JAVA TUTORIAL

access to multiple databases. You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type. Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

### Type 4 – 100% Pure Java

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.



This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can

## JAVA TUTORIAL

be downloaded dynamically. MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

### Which Driver should be Used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

### URL Formulation:

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded **DriverManager.getConnection()** methods –

- `getConnection(String url)`
- `getConnection(String url, Properties prop)`
- `getConnection(String url, String user, String password)`

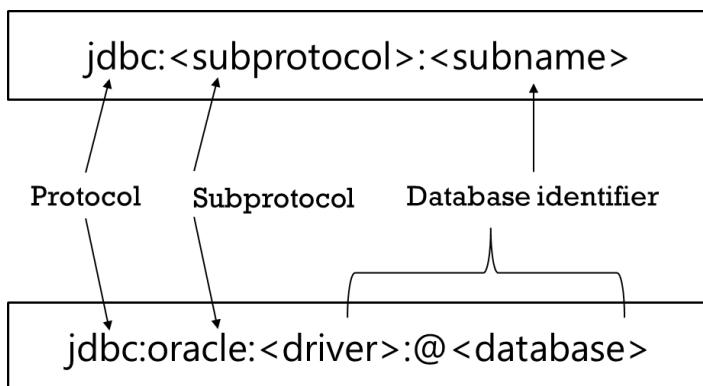
Here each form requires a database **URL**. A database URL is an address that points to your database.

**JAVA TUTORIAL**

Formulating a database URL is where most of the problems associated with establishing a connection occurs. Following table lists down the popular JDBC driver names and database URL.

A JDBC driver uses a JDBC URL to identify and connect to a particular database. These URLs are generally of the form:

`jdbc:<subprotocol>:<subname>` or `jdbc:driver:datasasename`



The actual standard is quite fluid, however, as different databases require different information to connect successfully. For example, the Oracle JDBC-Thin driver uses a URL of the form:

`jdbc:oracle:thin:@site:port:database`

while the JDBC-ODBC Bridge uses:

`jdbc:odbc:datasource:odboptions`

## JAVA TUTORIAL

The only requirement is that a driver be able to recognize its own URLs. The method `DriverManager.getConnection` establishes a database connection. This method requires a database URL, which varies depending on your DBMS. The following are some examples of database URLs:

1. MySQL: `jdbc:mysql://localhost:3306/`, where `localhost` is the name of the server hosting your database, and `3306` is the port number
2. Java DB: `jdbc:derby:testdb;create=true`, where `testdb` is the name of the database to connect to, and `create=true` instructs the DBMS to create the database.
3. **Note:** This URL establishes a database connection with the Java DB Embedded Driver. Java DB also includes a Network Client Driver, which uses a different URL.

This method specifies the username and password required to access the DBMS with a `Properties` object.

### **Note:**

- Typically, in the database URL, you also specify the name of an existing database to which you want to connect.

For example, the URL

`jdbc:mysql://localhost:3306/mysql`

represents the database URL for the MySQL database named `mysql`. The samples in this tutorial use a URL that does not specify a specific database because the samples create a new database.

## JAVA TUTORIAL

- In previous versions of JDBC, to obtain a connection, you first had to initialize your JDBC driver by calling the method Class.forName. This methods required an object of type java.sql.Driver. Each JDBC driver contains one or more classes that implements the interface java.sql.Driver. The drivers for Java DB are
  - org.apache.derby.jdbc.EmbeddedDriver and
  - org.apache.derby.jdbc.ClientDriver, and
  - the one for MySQL Connector/J is *com.mysql.cj.jdbc.Driver*.

Any JDBC 4.0 drivers that are found in your class path are automatically loaded. (However, you must manually load any drivers prior to JDBC 4.0 with the method Class.forName()). The method returns a Connection object, which represents a connection with the DBMS or a specific database. Query the database through this object.

### Specifying Database Connection URLs

A database connection URL is a string that your DBMS JDBC driver uses to connect to a database. It can contain information such as where to search for the database, the name of the database to connect to, and configuration properties. The exact syntax of a database connection URL is specified by your DBMS.

### Java DB Database Connection URLs

The following is the database connection URL syntax for Java DB:

## JAVA TUTORIAL

`jdbc:derby:[subsubprotocol:][databaseName][;attribute=value]*`

- *subsubprotocol* specifies where Java DB should search for the database, either in a directory, in memory, in a class path, or in a JAR file. It is typically omitted.
- *databaseName* is the name of the database to connect to.
- *attribute=value* represents an optional, semicolon-separated list of attributes. These attributes enable you to instruct Java DB to perform various tasks, including the following:
  - Create the database specified in the connection URL.
  - Encrypt the database specified in the connection URL.
  - Specify directories to store logging and trace information.
  - Specify a user name and password to connect to the database.

## MySQL Connector/J Database URL

The following is the database connection URL syntax for MySQL Connector/J:

`jdbc:mysql://[host][,failoverhost...][:port]/[database]  
[?propertyName1][=PropertyValue1]  
[&propertyName2][=PropertyValue2]...`

- *host:port* is the host name and port number of the computer hosting your database. If not specified, the default values of *host* and *port* are 127.0.0.1 and 3306, respectively.

## JAVA TUTORIAL

- *database* is the name of the database to connect to. If not specified, a connection is made with no default database.
- *failover* is the name of a standby database (MySQL Connector/J supports failover).
- *propertyName=propertyValue* represents an optional, ampersand-separated list of properties. These attributes enable you to instruct MySQL Connector/J to perform various tasks

Loading the driver or drivers you want to use is very simple and involves just one line of code. If, for example, you want to use the JDBC-ODBC Bridge driver, the following code will load it:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

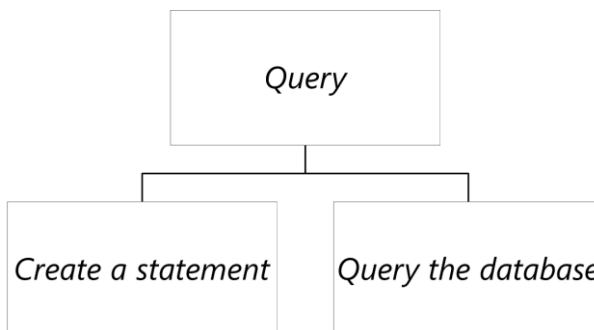
Your driver documentation will give you the class name to use. Class.forName will automatically register the driver with the DriverManager. When you have loaded a driver, it is available for making a connection with a DBMS.

The java.sql.Connection object, which encapsulates a single connection to a particular database, forms the basis of all JDBC data-handling code. The DriverManager.getConnection( ) method creates a connection:

```
Connection con = DriverManager.getConnection("url", "user",  
"password");
```

**JAVA TUTORIAL**

You pass three arguments to `getConnection()`: a JDBC URL, a database username, and a password. For databases that don't require explicit logins, the user and password strings should be left blank. When the method is called, the `DriverManager` queries each registered driver, asking if it understands the URL. If a driver recognizes the URL, it returns a `Connection` object. Because the `getConnection()` method checks each driver in turn, you should avoid loading more drivers than are necessary for your application.

*Stage 2: Query***Statements:**

A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate `ResultSet` objects, which is a table of data representing a database result set. You need a `Connection` object to create a Statement object.

For example, `CoffeesTables.viewTable` creates a Statement object with the following code:

## JAVA TUTORIAL

```
stmt = con.createStatement();
```

There are three different kinds of statements:

- Statement: Used to implement simple SQL statements with no parameters.
- PreparedStatement: (Extends Statement.) Used for precompiling SQL statements that might contain input parameters.
- CallableStatement: (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters.

Once a connection to a particular database is established, that connection can be used to send SQL statements. A Statement object is created with the Connection method createStatement, as in the following code fragment:

```
Statement stmt = conn.createStatement();
```

The SQL statement that will be sent to the database is supplied as the argument to one of the execute methods on a Statement object.

This is demonstrated in the following example, which uses the method executeQuery:

```
ResultSet rset = stmt.executeQuery("SELECT a, b, c FROM  
Table2");
```

**JAVA TUTORIAL**

The variable `rset` references a result set discussed in the following sections.

As mentioned earlier, the `Statement` interface provides three different methods for executing SQL statements: `executeQuery`, `executeUpdate`, and `execute`. The correct method to use is determined by what the SQL statement produces.

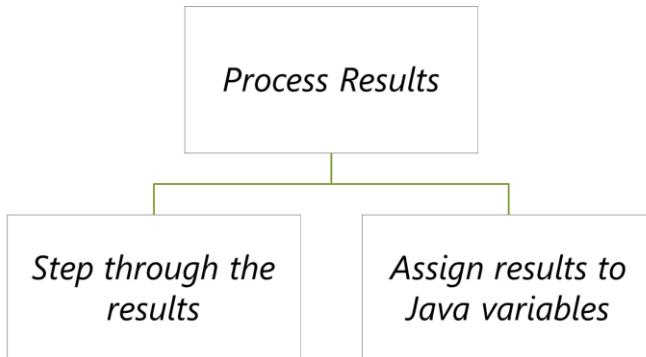
The method `executeQuery` is designed for statements that produce a single result set, such as `SELECT` statements.

The method `executeUpdate` is used to execute `INSERT`, `UPDATE`, or `DELETE` statements and also SQL DDL (Data Definition Language) statements like `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE`. The effect of an `INSERT`, `UPDATE`, or `DELETE` statement is a modification of one or more columns in zero or more rows in a table. The return value of `executeUpdate` is an integer (referred to as the update count) that indicates the number of rows that were affected. For statements such as `CREATE TABLE` or `DROP TABLE`, which do not operate on rows, the return value of `executeUpdate` is always zero.

The method `execute` is used to execute statements that return more than one result set, more than one update count, or a combination of the two.

## JAVA TUTORIAL

### *Stage 3: Process the Results*



### ResultSet Object:

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are type, concurrency, and cursor *holdability*.

### ResultSet Types

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object.

The sensitivity of a ResultSet object is determined by one of three different ResultSet types:

**JAVA TUTORIAL**

- **TYPE\_FORWARD\_ONLY:** The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE\_SCROLL\_INSENSITIVE:** The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE\_SCROLL\_SENSITIVE:** The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default *ResultSet* type is **TYPE\_FORWARD\_ONLY**.

**Note:** Not all databases and JDBC drivers support all *ResultSet* types.

Method `DatabaseMetaData.supportsResultSetType` returns true if the specified *ResultSet* type is supported and false otherwise.

**ResultSet Concurrency**

## JAVA TUTORIAL

The concurrency of a ResultSet object determines what level of update functionality is supported.

There are two concurrency levels:

- CONCUR\_READ\_ONLY: The ResultSet object cannot be updated using the ResultSet interface.
- CONCUR\_UPDATABLE: The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR\_READ\_ONLY.

**Note:** Not all JDBC drivers and databases support concurrency.

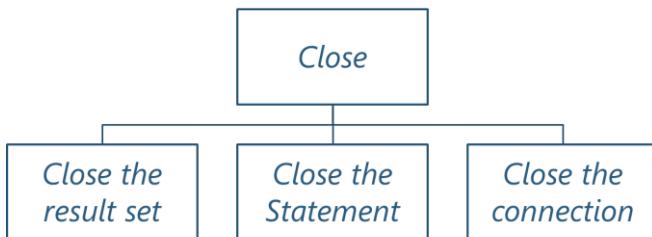
DatabaseMetaData supports ResultSetConcurrency returns true if the specified concurrency level is supported by the driver and false otherwise.

### Cursor Holdability

Connection.commit calling can close the ResultSet objects that have been created during the current transaction. In some cases, however, this may not be the desired behavior. The ResultSet property *holdability* gives the application control over whether ResultSet objects (cursors) are closed when commit is called. The following ResultSet constants may be supplied to the Connection methods createStatement, prepareStatement, and prepareCall:

**JAVA TUTORIAL**

- **HOLD\_CURSORS\_OVER\_COMMIT:** ResultSet cursor s are not closed; they are *holdable*: they are held open when the method commit is called. Holdable cursors might be ideal if your application uses mostly read-only ResultSet objects.
- **CLOSE\_CURSORS\_AT\_COMMIT:** ResultSet objects (cursors) are closed when the commit method is called. Closing cursors when this method is called can result in better performance for some applications.
- A ResultSet object maintains a cursor, which points to its current row of data. The cursor moves down one row each time the method next is called. When a ResultSet object is first created, the cursor is positioned before the first row, so the first call to the next method puts the cursor on the first row, making it the current row. ResultSet rows can be retrieved in sequence from top to bottom as the cursor moves down one row with each successive call to the method next.
- When a cursor is positioned on a row in a ResultSet object (not before the first row or after the last row), that row becomes the current row. This means that any methods called while the cursor is positioned on that row will operate on values in that row (methods such as getXXX).
- A cursor remains valid until the ResultSet object or its parent Statement object is closed.

**JAVA TUTORIAL***Stage 4: Close*

When a connection is in auto-commit mode, the statements being executed within it are committed or rolled back when they are completed. A statement is considered complete when it has been executed and all its results have been returned. For the method `executeQuery`, which returns one result set, the statement is completed when all the rows of the `ResultSet` object have been retrieved. For the method `executeUpdate`, a statement is completed when it is executed. In the rare cases where the method `execute` is called, however, a statement is not complete until all of the result sets or update counts it generated have been retrieved.

Statement objects will be closed automatically by the Java garbage collector. Nevertheless, it is recommended as good programming practice that they be closed explicitly when they are no longer needed. This frees DBMS resources immediately and helps avoid potential memory problems.

**JAVA TUTORIAL**

The same connection object can be used to execute multiple statements and retrieve many result sets. However, once all the work with the database is over, it is a good programming practice to close the connection explicitly. If this is not closed, after a particular timeout period defined at the database, the connection is automatically closed. Nevertheless, this would mean that till the timeout, this connection is not available to any other users of the database. Hence, explicit closing of the connection is recommended.

### *The DatabaseMetaData Object*

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

Commonly used methods of DatabaseMetaData interface

- **public String getDriverName()throws SQLException:** it returns the name of the JDBC driver.
- **public String getDriverVersion()throws SQLException:** it returns the version number of the JDBC driver.
- **public String getUserName()throws SQLException:** it returns the username of the database.
- **public String getDatabaseProductName()throws SQLException:** it returns the product name of the database.

**JAVA TUTORIAL**

- **public String getDatabaseProductVersion()throws SQLException:** it returns the product version of the database.
- **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types) throws SQLException:** it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

To obtain DatabaseMetaData object,

```
DatabaseMetaData dbmd = conn.getMetaData();
```

This interface is implemented by driver vendors to let users know the capabilities of a Database Management System (DBMS) in combination with the driver based on JDBC technology ("JDBC driver") that is used with it. Different relational DBMSs often support different features, implement features in different ways, and use different data types. In addition, a driver may implement a feature on top of what the DBMS offers. Information returned by methods in this interface applies to the capabilities of a particular driver and a particular DBMS working together.

A user for this interface is commonly a tool that needs to discover how to deal with the underlying DBMS. This is especially true for applications that are intended to be used with more than one DBMS.

For example:

- **getURL():** Returns the URL for the DBMS

**JAVA TUTORIAL**

- `getSQLKeywords()`: Retrieves a comma-separated list of all of this database's SQL keywords that are NOT also SQL92 keywords.
- `supportsTransactions()`: Retrieves whether this database supports transactions. If not, invoking the method `commit` is no use, and the isolation level is `TRANSACTION_NONE`.
- `supportsSelectForUpdate()`: Retrieves whether this database supports `SELECT FOR UPDATE` statements.

## The *ResultSetMetaData* Object

### **ResultSetMetaData**

In JDBC, you use the `ResultSet.getMetaData()` method to return a `ResultSetMetaData` object, which describes the data coming back from a database query. This object can be used to find out about the types and properties of the columns in your `ResultSet`.

<i>Method</i>	<i>Description</i>
<code>getColumnCount()</code>	Retrieves the number of columns in the current <code>ResultSet</code> object.
<code>getColumnLabel()</code>	Retrieves the suggested name of the column for use.
<code>getColumnName()</code>	Retrieves the name of the column.
<code>getTableName()</code>	Retrieves the name of the table.

`ResultSetMetaData` is an interface in `java.sql` package of JDBC API which is used to get the metadata about

## JAVA TUTORIAL

a *ResultSet* object. Whenever you query the database using SELECT statement, the result will be stored in a *ResultSet* object.

Every *ResultSet* object is associated with one *ResultSetMetaData* object. This object will have all the meta data about a *ResultSet* object like schema name, table name, number of columns, column name, datatype of a column etc.

You can get this *ResultSetMetaData* object using *getMetaData()* method of *ResultSet*.

An object that can be used to get information about the types and properties of the columns in a *ResultSet* object. The following code fragment creates the *ResultSet* object rs, creates the *ResultSetMetaData* object rsmd, and uses rsmd to find out how many columns rs has and whether the first column in rs can be used in a WHERE clause.

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");
```

```
ResultSetMetaData rsmd = rs.getMetaData();  
int numberOfColumns = rsmd.getColumnCount();  
boolean b = rsmd.isSearchable(1);
```

### Example

The example on the slide shows how to use a *ResultSetMetaData* object to determine the following information about the *ResultSet*:

The number of columns in the *ResultSet*.

## JAVA TUTORIAL

The name of each column

The American National Standards Institute (ANSI) SQL type for each column

### **java.sql.Types**

The `java.sql.Types` class defines constants that are used to identify ANSI SQL types. `ResultSetMetaData.getColumnType()` returns an integer value that corresponds to one of these constants.

## Mapping Database Types to Java Types

Because data types in SQL and data types in the Java programming language are not identical, there needs to be some mechanism for transferring data between an application using Java types and a database using SQL types. In order to transfer data between a database and an application written in the Java programming language, the JDBC API provides three sets of methods:

1. Methods on the `ResultSet` class for retrieving SQL SELECT results as Java types
2. Methods on the `PreparedStatement` class for sending Java types as SQL statement parameters
3. Methods on the `CallableStatement` class for retrieving SQL OUT parameters as Java types

SHOIB MOHAMMAD

9840195749

## JAVA TUTORIAL

JDBC Type	Java Type
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double

BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	Clob
BLOB	Blob
ARRAY	Array
DISTINCT	mapping of underlying type
STRUCT	Struct
REF	Ref
JAVA_OBJECT	underlying Java class

## JAVA TUTORIAL

JDBC defines a standard mapping from the JDBC database types to Java types. For example, a JDBC INTEGER is normally mapped to a Java int. This supports a simple interface for reading and writing JDBC values as simple Java types.

The methods

*ResultSet.getObject and CallableStatement.getObject*

retrieve a value as a Java Object. Since Object is the base class for all Java objects, an instance of any Java class can be retrieved as an instance of Object.

However, the following Java types are built-in "primitive" types and are therefore not instances of the class

Object: boolean, char, byte, short, int, long, float, and double.

As a result, these types cannot be retrieved by *getObject* methods.

However, each of these primitive types has a corresponding class that serves as a wrapper. Instances of these classes are objects, which means that they can be retrieved with the methods -

*ResultSet.getObject and CallableStatement.getObject.*

## The *PreparedStatement* Object

Prepared Statements

PreparedStatement is inherited from Statement; the difference is that a PreparedStatement holds precompiled SQL statements.

## JAVA TUTORIAL

If you execute a Statement object many times, its SQL statement is compiled each time. PreparedStatement is more efficient because its SQL statement is compiled only once, when you first prepare the PreparedStatement. After that, each time you execute the SQL statement in the PreparedStatement, the SQL statement does not have to be recompiled.

Therefore, if you need to execute the same SQL statement several times within an application, it is more efficient to use PreparedStatement than Statement.

### Create a PreparedStatement

- Register the driver and create the database connection

```
PreparedStatement pstmt =  
    conn.prepareStatement("update ACME_RENTALS  
    set STATUS = ? where RENTAL_ID = ?");
```

- Create the prepared statement, identifying variables with a question mark (?)

```
PreparedStatement pstmt =  
    conn.prepareStatement("select STATUS from  
    ACME_RENTALS where RENTAL_ID = ?");
```

### ***PreparedStatement Parameters***

## JAVA TUTORIAL

A *PreparedStatement* does not have to execute exactly the same query each time. You can specify parameters in the *PreparedStatement* SQL string and supply the actual values for these parameters when the statement is executed. The following slide shows how to supply parameters and execute a *PreparedStatement*.

Even for creating *PreparedStatement* object, the initial steps of registering the driver and creating a connection object remain the same. Once the connection object is obtained, the *prepareStatement* method is called on it to obtain the *PreparedStatement* object. However, in this case, while creating it, itself, the SQL statement is provided as a parameter to the method. The variable portions of the SQL statement are provided as a question mark (?) so that the values can be supplied dynamically before execution of the statement.

### Specifying Values for the Bind Variables

You use the *PreparedStatement.setXXX()* methods to supply values for the variables in a prepared statement. There is one *setXXX()* method for each Java type: *setString()*, *setInt()*, and so on.

You must use the *setXXX()* method that is compatible with the SQL type of the variable. In the example on the slide, the first variable is updating a VARCHAR column, so we need to use *setString()* to supply a value for the variable. You can use *setObject()* with any variable type.

## JAVA TUTORIAL

Each variable has an index. The index of the first variable in the prepared statement is 1, the index of the second is 2, and so on. If there is only one variable, its index is one. The index of a variable is passed to the setXXX() method.

### Closing a Prepared Statement

If you close a prepared statement, you will have to prepare it again.

## The *CallableStatement* Object

The way to access stored procedures using JDBC is through the *CallableStatement* class which is inherited from the *PreparedStatement* class.

*CallableStatement* is like *PreparedStatement* in that you can specify parameters using the question mark (?) notation, but it contains no SQL statements.

Both functions and procedures take parameters represented by identifiers. A function executes some procedural logic and it returns a value that can be any data type supported by the database. The parameters supplied to the function do not change after the function is executed.

A procedure executes some procedural logic but does not return any value. However, some of the parameters supplied to the procedure may have their values changed after the procedure is executed.

**Note:** Calling a stored procedure is the same whether the stored procedure was written originally in Java or in any other language

**JAVA TUTORIAL**

supported by the database, such as PL/SQL. Indeed, a stored procedure written in Java appears to the programmer as a PL/SQL stored procedure.

***Creating a Callable Statement***

```
CallableStatement cstmt =  
    conn.prepareCall("{call " +  
        ADDITEM +  
        "(?, ?, ?)}");  
    cstmt.registerOutParameter(2, Types.INTEGER);  
    cstmt.registerOutParameter(3, Types.DOUBLE);
```

First you need an active connection to the database in order to obtain a CallableStatement object.

Next, you create a CallableStatement object using the prepareCall() method of the Connection class. This method typically takes a string as an argument. The syntax for the string has two forms. The first form includes a result parameter and the second form does not:

```
{? = call proc (...) } // A result is returned into a variable  
{call proc (...) } // Does not return a result
```

In the example in the slide, the second form is used, where the stored procedure in question is ADDITEM.

Note that the parameters to the stored procedures are specified using the question mark notation used earlier in

## JAVA TUTORIAL

PreparedStatement. You must register the data type of the parameters using the registerOutParameter() method of CallableStatement if you expect a return value, or if the procedure is going to modify a variable (also known as an OUT variable). In the example in the slide, the second and third parameters are going to be computed by the stored procedure, whereas the first parameter is an input (the input is specified in the next slide). Parameters are referred to sequentially, by number. The first parameter is 1.

To specify the data type of each OUT variable, you use parameter types from the Types class. When the stored procedure successfully returns, the values can be retrieved from the CallableStatement object.

### How to Execute a Callable Statement

There are three steps in executing the stored procedure after you have registered the types of the OUT variables:

1. Set the IN parameters.

Use the setXXX() methods to supply values for the IN parameters. There is one setXXX() method for each Java type: setString(), setInt(), and so on. You must use the setXXX() method that is compatible with the SQL type of the variable. You can use setObject() with any variable type. Each variable has an index. The index of the first variable in the callable statement is 1, the index of the second is 2, and so on. If there is only one variable, its index is 1.

2. Execute the call to the stored procedure.

**JAVA TUTORIAL**

Execute the procedure using the execute() method.

3. Get the OUT parameters.

Once the procedure is completed, you retrieve OUT variables, if any, using the getXXX() methods. Note that these methods must match the types you registered in the previous slide.

## Using *Transactions*

With JDBC, database transactions are managed by the Connection object. When you create a Connection object, it is in autocommit mode, meaning that each statement is committed after it is executed.

You can change the connection's autocommit mode at any time by calling setAutoCommit(). Here is a full description of autocommit mode:

- If a connection is in autocommit mode, all its SQL statements will be executed and committed as individual transactions.
- If a statement returns a result set, the statement completes when the last row of the result set has been retrieved, or the result set has been closed.
- If autocommit mode has been disabled, its SQL statements are grouped into transactions, which must be terminated by calling either *commit()* or *rollback()*. *commit()* makes permanent all changes since the previous commit or rollback and releases any database locks held by the connection.

## JAVA TUTORIAL

*rollback()* drops all changes since the previous commit or rollback and releases any database locks. *commit()* and *rollback()* should only be called when in non-autocommit mode.

### Transaction Isolation:

Call `con.setTransactionIsolation(int level)` to set the transaction isolation level. The transaction isolation level controls what happens when concurrent transactions affect the same data.

### Dirty Reads

- A *dirty read* occurs when a transaction can see uncommitted changes to a row. That is, changes made by a transaction are visible before the transaction is committed. The danger is that the change may be rolled back.

### Non Repeatable Reads

- A *non-repeatable read* is when a row is read twice within a transaction and gets different results.

1. Transaction A reads a row.
2. Transaction B modifies the same row and commits.
3. Transaction A reads the row again, getting a different result than the first time.
  1. Note that this could happen even if dirty reads are prevented.

**JAVA TUTORIAL**

2. In the example above, Transaction B committed the change, so the second read by A was not a dirty read.

**Phantom Reads**

A *phantom read* is caused when a transaction can read a row inserted by another transaction.

Transaction A does a SELECT with a WHERE clause and gets some collection of rows. Transaction B inserts a row that satisfies the WHERE clause.

Transaction A performs the SELECT again and reads the inserted row.

*Isolation Levels:*

`TRANSACTION_NONE` - the driver does not support transactions (this is not an SQL99 transaction isolation level).

`TRANSACTION_READ_UNCOMMITTED` - Data modified by one transaction can be seen outside the transaction before it is committed. This allows dirty reads, non-repeatable reads, and phantom reads.

`TRANSACTION_READ_COMMITTED` - Data modified by a transaction is not visible outside the transaction until committed. Prevents dirty reads, but still allows non-repeatable and phantom reads.

`TRANSACTION_REPEATABLE_READ` - Prevents dirty and non-repeatable reads. Phantom reads can still occur.

**JAVA TUTORIAL**

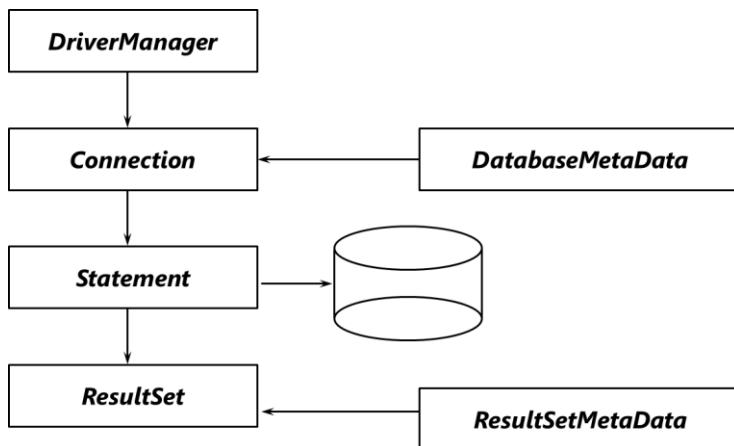
TRANSACTION\_SERIALIZABLE - Prevents dirty, non-repeatable, and phantom reads.

The interface *Connection* includes five values that represent the transaction isolation levels you can use in JDBC:

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	<i>Not supported</i>	<i>Not applicable</i>	<i>Not applicable</i>	<i>Not applicable</i>
TRANSACTION_READ_COMMITTED	<i>Supported</i>	<i>Prevented</i>	<i>Allowed</i>	<i>Allowed</i>
TRANSACTION_READ_UNCOMMITTED	<i>Supported</i>	<i>Allowed</i>	<i>Allowed</i>	<i>Allowed</i>
TRANSACTION_REPEATABLE_READ	<i>Supported</i>	<i>Prevented</i>	<i>Prevented</i>	<i>Allowed</i>
TRANSACTION_SERIALIZABLE	<i>Supported</i>	<i>Prevented</i>	<i>Prevented</i>	<i>Prevented</i>

## JAVA TUTORIAL

## Summary of JDBC Classes



### DriverManager

DriverManager provides access to registered JDBC drivers. DriverManager hands out connections to a specified data source through its `getConnection()` method.

### Connection

The Connection class is provided by the JDBC driver, as are all subsequent classes mentioned. A Connection object represents a session with a database and is used to create a Statement object, using `Connection.createStatement()`.

### Statement

## JAVA TUTORIAL

The Statement class executes SQL statements. For example, queries can be executed using the executeQuery() method and the results are wrapped up in a ResultSet object.

### ResultSet

JDBC returns the results of a query in a ResultSet object. A ResultSet object maintains a cursor pointing to its current row of data. The next() method moves the cursor to the next row. The ResultSet class has getXXX() methods to retrieve the columns in the current row.

### DatabaseMetaData and ResultSetMetaData

The DatabaseMetaData and ResultSetMetaData classes return metadata about the database and ResultSet, respectively. Call getMetaData() on the Connection object or the ResultSet object.