Unit 1

Overview of Java concepts: Classes, Objects, Inheritance, Polymorphism and Abstraction. Concepts of interface and Object up casting and down casting. Exception Handling. Concept of Static blocks. J2EE and J2SE, Why J2EE? Client/Server, 3 tier and N tier Systems.

Java Programming

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Java is −

- **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** − With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** − Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

- **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** − Java is designed for the distributed environment of the internet.
- **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java

James Gosling initiated Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called 'Oak' after an oak tree that stood outside Gosling's office, also went by the name 'Green' and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May, 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Required Software

For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

You will also need the following softwares −

- Linux 7.1 or Windows xp/7/8 operating system
- Java JDK 8
- Microsoft Notepad or any other text editor

Applications of Java Programming

The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2EE for Enterprise Applications, J2ME for Mobile Applications.

The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** − Java is designed for the distributed environment of the internet.
- **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Local Environment Setup

If you are still willing to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Following are the steps to set up the environment.

Java SE is freely available from the website. You can download a version based on your operating system.

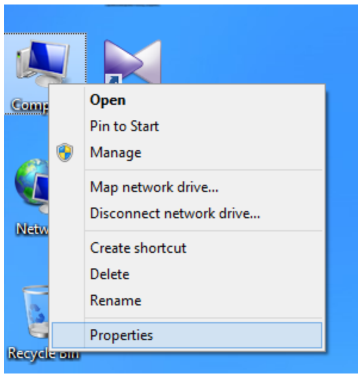
Follow the instructions to download Java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you will need to set environment variables to point to correct installation directories −

Setting Up the Path for Windows

Assuming you have installed Java in *c:\Program Files\java\jdk* directory −

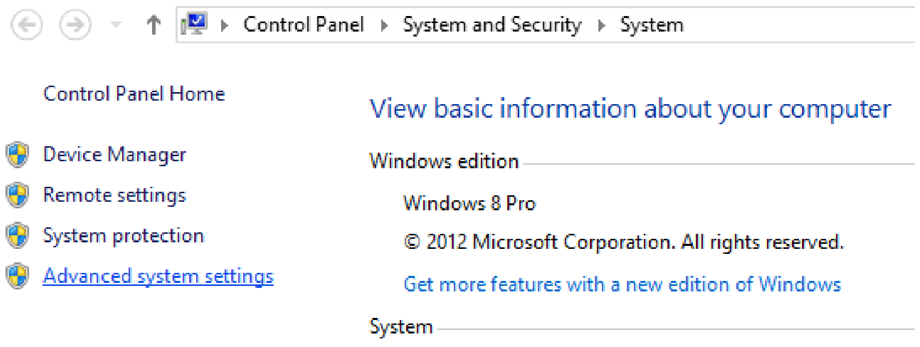
- Right-click on 'My Computer' and select 'Properties'.
- Click the 'Environment variables' button under the 'Advanced' tab.
- Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting PATH Environment



Right click on My Computer and select properties

Click on Advanced system settings



View basic information about your computer

Windows edition

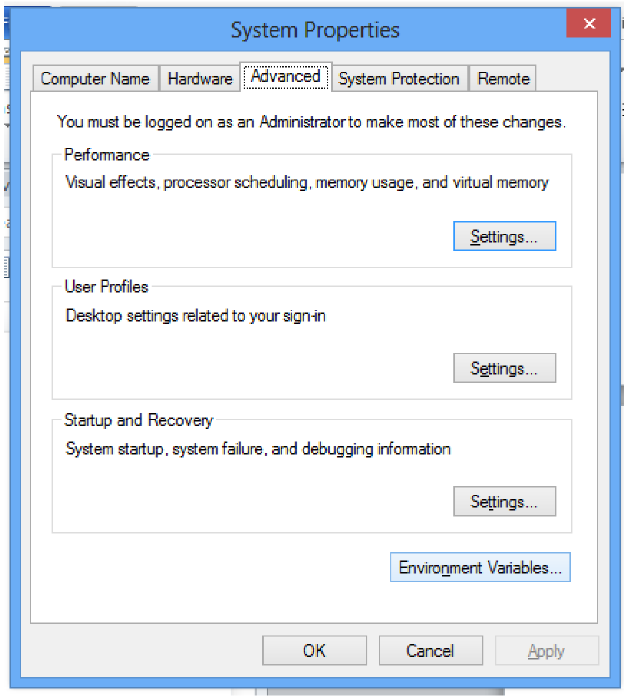
Windows 8 Pro

© 2012 Microsoft Corporation. All rights reserved.

Get more features with a new edition of Windows

System

Click on Advanced tab and click on Environment Variables


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...

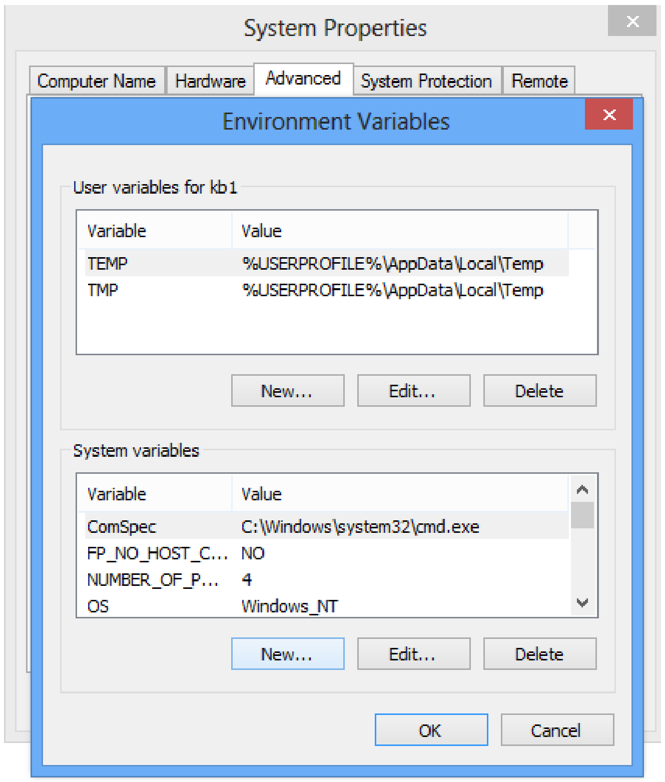
User Profiles
Desktop settings related to your sign-in
Settings...

Startup and Recovery
System startup, system failure, and debugging information
Settings...

Environment Variables...

OK | Cancel | Apply

Under system properties, check if there is any variable named PATH, if not create a new variable


System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

Environment Variables

User variables for kb1

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

New... | Edit... | Delete

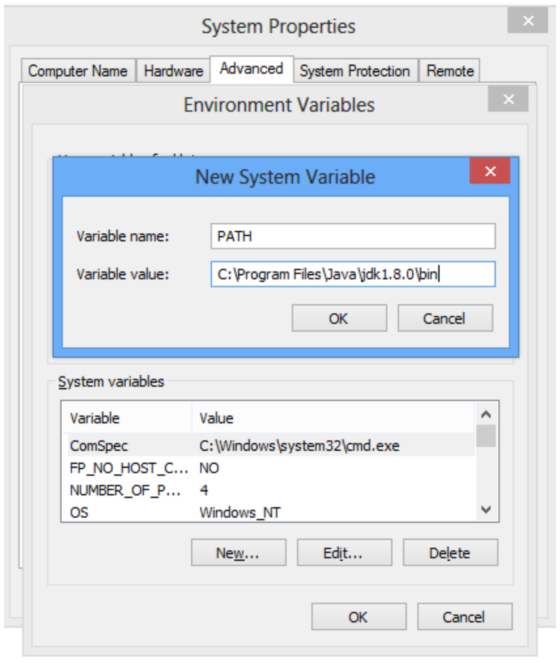
System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	4
OS	Windows_NT

New... | Edit... | Delete

OK | Cancel

Enter variable name as PATH and variable value as java installed bin directory path in your system



Variable name: PATH

Variable value: C:\Program Files\Java\jdk1.8.0\bin

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	4
OS	Windows_NT

Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation, if you have trouble doing this.

Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH = /path/to/java:$PATH'

Popular Java Editors

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following −

- **Notepad** − On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
- **Netbeans** − A Java IDE that is open-source and free which can be downloaded from **https://www.netbeans.org/index.html**.
- **Eclipse** − A Java IDE developed by the eclipse open-source community and can be downloaded from **https://www.eclipse.org/**.

Java - Basic Syntax

- **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behavior such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- **Methods** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

```
public class MyFirstJavaProgram {

   /* This is my first java program.
    * This will print 'Hello World' as the output
    */

   public static void main(String []args)
   {
      System.out.println("Hello World"); // prints Hello World
   }
}
```

Let's look at how to save the file, compile, and run the program. Please follow the subsequent steps −

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

Output

```
C:\> javac MyFirstJavaProgram.java
C:\> java MyFirstJavaProgram
Hello World
```

Basic Syntax

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity** − Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names** − For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

  Example: *class MyFirstJavaClass*

- **Method Names** − All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

  Example: *public void myMethodName()*

- **Program File Name** − Name of the program file should exactly match the class name.

  When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).

  But please make a note that in case you do not have a public class present in the file then file name can be different than class name. It is also not mandatory to have a public class in the file.

  Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*

- **public static void main(String args[])** − Java program processing starts from the main() method which is a mandatory part of every Java program.

Java Identifiers

All Java components require names. Names used for classes, variables, and methods are called **identifiers**.

In Java, there are several points to remember about identifiers. They are as follows −

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly, identifiers are case sensitive.

- Examples of legal identifiers: age, $salary, _value, __1_value.
- Examples of illegal identifiers: 123abc, -salary.

Java Modifiers

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers −

- **Access Modifiers** − default, public , protected, private
- **Non-access Modifiers** − final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

Java Variables

Following are the types of variables in Java −

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static Variables)

Java Arrays

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap. We will look into how to declare, construct, and initialize in the upcoming chapters.

Java Enums

Enums were introduced in Java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium, and large. This would make sure that it would not allow anyone to order any size other than small, medium, or large.

```
class FreshJuice {
   enum FreshJuiceSize{ SMALL, MEDIUM, LARGE }
   FreshJuiceSize size;
}

public class FreshJuiceTest {

   public static void main(String args[]) {
      FreshJuice juice = new FreshJuice();
      juice.size = FreshJuice.FreshJuiceSize.MEDIUM ;
      System.out.println("Size: " + juice.size);
   }
}
```

The above example will produce the following result −

Output

```
Size: MEDIUM
```

Note − Enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Java Keywords

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

Comments in Java

Java supports single-line and multi-line comments very similar to C and C++. All characters available inside any comment are ignored by Java compiler.

Example

Live Demo

```
public class MyFirstJavaProgram {

   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */

   public static void main(String []args) {
      // This is an example of single line comment
      /* This is also an example of single line comment. */
      System.out.println("Hello World");
   }
}
```

Output

```
Hello World
```

Using Blank Lines

A line containing only white space, possibly with a comment, is known as a blank line, and Java totally ignores it.

Inheritance

In Java, classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

This concept allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the **superclass** and the derived class is called the **subclass**.

Interfaces

In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

An interface defines the methods, a deriving class (subclass) should use. But the implementation of the methods is totally up to the subclass.

Java is an Object-Oriented Language. As a language that has the Object-Oriented feature, Java supports the following fundamental concepts −

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method

- Message Passing

In this chapter, we will look into the concepts - Classes and Objects.

- **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.
- **Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Objects in Java

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If you compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

A class is a blueprint from which individual objects are created.

Following is a sample of a class.

Example

```
public class Dog {
   String breed;
   int age;
   String color;

   void barking() {
   }

   void hungry() {
   }

   void sleeping() {
   }
}
```

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor −

Example

```
public class Puppy {
   public Puppy() {
   }

   public Puppy(String name) {
      // This constructor has one parameter, name.
   }
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Note − We have two different types of constructors. We are going to discuss constructors in detail in the subsequent chapters.

Creating an Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

- **Declaration** − A variable declaration with a variable name with an object type.
- **Instantiation** − The 'new' keyword is used to create the object.
- **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object −

Example

Live Demo

```
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }

   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

If we compile and run the above program, then it will produce the following result −

Output

```
Passed Name is :tommy
```

Accessing Instance Variables and Methods

Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path −

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example

This example explains how to access instance variables and methods of a class.

Live Demo

```
public class Puppy {
   int puppyAge;

   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Name chosen is :" + name );
   }

   public void setAge( int age ) {
      puppyAge = age;
   }

   public int getAge( ) {
      System.out.println("Puppy's age is :" + puppyAge );
      return puppyAge;
   }

   public static void main(String []args) {
      /* Object creation */
      Puppy myPuppy = new Puppy( "tommy" );

      /* Call class method to set puppy's age */
      myPuppy.setAge( 2 );

      /* Call another class method to get puppy's age */
      myPuppy.getAge( );

      /* You can access instance variable as follows as well */
      System.out.println("Variable Value :" + myPuppy.puppyAge );
   }
}
```

If we compile and run the above program, then it will produce the following result −

Output

```
Name chosen is :tommy
Puppy's age is :2
Variable Value :2
```

Source File Declaration Rules

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non-public classes.
- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example: the class name is *public class Employee{}* then the source file should be as Employee.java.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.

- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.

Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import Statements

In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory java_installation/java/io −

```
import java.io.*;
```

A Simple Case Study

For our case study, we will be creating two classes. They are Employee and EmployeeTest.

First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.

The Employee class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

Example

```
import java.io.*;
public class Employee {

   String name;
   int age;
   String designation;
   double salary;
   // This is the constructor of the class Employee
   public Employee(String name) {
      this.name = name;
   }

   // Assign the age of the Employee  to the variable age.
   public void empAge(int empAge) {
      age = empAge;
   }

   /* Assign the designation to the variable designation.*/
   public void empDesignation(String empDesig) {
      designation = empDesig;
   }

   /* Assign the salary to the variable       salary.*/
   public void empSalary(double empSalary) {
      salary = empSalary;
   }

   /* Print the Employee details */
   public void printEmployee() {
      System.out.println("Name:"+ name );
      System.out.println("Age:" + age );
      System.out.println("Designation:" + designation );
      System.out.println("Salary:" + salary);
   }
}
```

As mentioned previously in this tutorial, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.

Following is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file.

```
import java.io.*;
public class EmployeeTest {

   public static void main(String args[]) {
      /* Create two objects using constructor */
      Employee empOne = new Employee("James Smith");
      Employee empTwo = new Employee("Mary Anne");

      // Invoking methods for each object created
      empOne.empAge(26);
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

      empTwo.empAge(21);
      empTwo.empDesignation("Software Engineer");
      empTwo.empSalary(500);
      empTwo.printEmployee();
   }
}
```

Now, compile both the classes and then run *EmployeeTest* to see the result as follows −

Output

```
C:\> javac Employee.java
C:\> javac EmployeeTest.java
C:\> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

Java - Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
   .....
   .....
}
class Sub extends Super {
   .....
   .....
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

Copy and paste the following program in a file with name My_Calculation.java

Example

Live Demo

```
class Calculation {
   int z;

   public void addition(int x, int y) {
      z = x + y;
      System.out.println("The sum of the given numbers:"+z);
   }

   public void Subtraction(int x, int y) {
      z = x - y;
      System.out.println("The difference between the given numbers:"+z);
   }
}

public class My_Calculation extends Calculation {
   public void multiplication(int x, int y) {
      z = x * y;
      System.out.println("The product of the given numbers:"+z);
   }

   public static void main(String args[]) {
      int a = 20, b = 10;
      My_Calculation demo = new My_Calculation();
      demo.addition(a, b);
      demo.Subtraction(a, b);
      demo.multiplication(a, b);
   }
}
```

Compile and execute the above code as shown below.

```
javac My_Calculation.java
java My_Calculation
```

After executing the program, it will produce the following result −

Output

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.

The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass My_Calculation.

```
Calculation demo = new My_Calculation();
demo.addition(a, b);
demo.Subtraction(a, b);
```

Note − A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable
super.method();
```

Sample Code

This section provides you a program that demonstrates the usage of the **super** keyword.

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name Sub_class.java.

Example

Live Demo

```
class Super_class {
   int num = 20;

   // display method of superclass
   public void display() {
      System.out.println("This is the display method of superclass");
   }
}

public class Sub_class extends Super_class {
   int num = 10;

   // display method of sub class
   public void display() {
      System.out.println("This is the display method of subclass");
   }

   public void my_method() {
      // Instantiating subclass
      Sub_class sub = new Sub_class();

      // Invoking the display() method of sub class
      sub.display();

      // Invoking the display() method of superclass
      super.display();

      // printing the value of variable num of subclass
      System.out.println("value  of  the  variable  named  num  in  sub  class:"+
sub.num);

      // printing the value of variable num of superclass
      System.out.println("value  of  the  variable  named  num  in  super  class:"+
super.num);
   }

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

Compile and execute the above code using the following syntax.

```
javac Super_Demo
java Super
```

On executing the program, you will get the following result −

Output

```
This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20
```

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

Example

Live Demo

```
class Superclass {
   int age;

   Superclass(int age) {
      this.age = age;
   }

   public void getAge() {
      System.out.println("The value of the variable named age in super class 
is: " +age);
   }
}

public class Subclass extends Superclass {
   Subclass(int age) {
      super(age);
   }

   public static void main(String argd[]) {
      Subclass s = new Subclass(24);
      s.getAge();
   }
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass
java Subclass
```

On executing the program, you will get the following result −

Output

```
The value of the variable named age in super class is: 24
```

Example

Live Demo
```
class Animal {
}

class Mammal extends Animal {
}

class Reptile extends Animal {
}

public class Dog extends Mammal {

   public static void main(String args[]) {
      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This will produce the following result −

Output

```
true
true
true
```

Since we have a good understanding of the **extends** keyword, let us look into how the **implements** keyword is used to get the IS-A relationship.

Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

Example

```
public interface Animal {
}

public class Mammal implements Animal {
}

public class Dog extends Mammal {
}
```

The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

Example

Live Demo

```
interface Animal{}
class Mammal implements Animal{}

public class Dog extends Mammal {

   public static void main(String args[]) {
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This will produce the following result −

Output

```
true
true
true
```

HAS-A relationship

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets look into an example −

Example

```
public class Vehicle{}
public class Speed{}

public class Van extends Vehicle {
   private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class, which makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So, basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

Types of Inheritance

There are various types of inheritance as demonstrated below.

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal −

Example

```
public class extends Animal, Mammal{}
```

However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance.

Java – Overriding

If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

Example

Let us look at an example.

Live Demo
```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
}

public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
   }
}
```

This will produce the following result −

Output
```
Animals can move
Dogs can walk and run
```

In the above example, you can see that even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

Consider the following example −

Example
Live Demo
```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      System.out.println("Dogs can walk and run");
   }
   public void bark() {
      System.out.println("Dogs can bark");
   }
}

public class TestDog {

   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
      b.bark();
   }
}
```

This will produce the following result −

Output

```
TestDog.java:26: error: cannot find symbol
      b.bark();
       ^
  symbol:   method bark()
  location: variable b of type Animal
1 error
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.

- Constructors cannot be overridden.

Using the super Keyword

When invoking a superclass version of an overridden method the **super** keyword is used.

Example

Live Demo

```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}

class Dog extends Animal {
   public void move() {
      super.move();   // invokes the super class method
      System.out.println("Dogs can walk and run");
   }
}

public class TestDog {

   public static void main(String args[]) {
      Animal b = new Dog();   // Animal reference but Dog object
      b.move();   // runs the method in Dog class
   }
}
```

This will produce the following result −

Output

```
Animals can move
Dogs can walk and run
```

Java – Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal −

Example

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

Example

```
/* File name : Employee.java */
public class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number) {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public void mailCheck() {
      System.out.println("Mailing  a  check  to  "  +  this.name  +  "  "  +
this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```

Now suppose we extend Employee class as follows −

```
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary; // Annual salary

   public Salary(String name, String address, int number, double salary) {
      super(name, address, number);
      setSalary(salary);
   }

   public void mailCheck() {
      System.out.println("Within mailCheck of Salary class ");
      System.out.println("Mailing check to " + getName()
      + " with salary " + salary);
   }
   public double getSalary() {
      return salary;
   }

   public void setSalary(double newSalary) {
      if(newSalary >= 0.0) {
         salary = newSalary;
      }
   }

   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

Now, you study the following program carefully and try to determine its output −

```
/* File name : VirtualDemo.java */
public class VirtualDemo {

   public static void main(String [] args) {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

This will produce the following result −

Output

```
Constructing an Employee
Constructing an Employee

Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Here, we instantiate two Salary objects. One using a Salary reference **s**, and the other using an Employee reference **e**.

While invoking *s.mailCheck()*, the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

mailCheck() on **e** is quite different because **e** is an Employee reference. When the compiler sees *e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time

Java - Abstraction

Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Example

This section provides you an example of the abstract class. To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

```
/* File name : Employee.java */
public abstract class Employee {
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number) {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }

   public double computePay() {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }

   public void mailCheck() {
      System.out.println("Mailing  a  check  to  "  +  this.name  +  "  "  +
this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
   }

   public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way −

```
/* File name : AbstractDemo.java */
public class AbstractDemo {

   public static void main(String [] args) {
      /* Following is not allowed and would raise error */
      Employee e = new Employee("George W.", "Houston, TX", 43);
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

When you compile the above class, it gives you the following error −

```
Employee.java:46: Employee is abstract; cannot be instantiated
      Employee e = new Employee("George W.", "Houston, TX", 43);
                   ^
1 error
```

Inheriting the Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way −

Example

```
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary;   // Annual salary
   
   public Salary(String name, String address, int number, double salary) {
      super(name, address, number);
      setSalary(salary);
   }
   
   public void mailCheck() {
      System.out.println("Within mailCheck of Salary class ");
      System.out.println("Mailing check to " + getName() + " with salary " + 
salary);
   }

   public double getSalary() {
      return salary;
   }
   
   public void setSalary(double newSalary) {
      if(newSalary >= 0.0) {
         salary = newSalary;
      }
   }
   
   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```
/* File name : AbstractDemo.java */
public class AbstractDemo {

   public static void main(String [] args) {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

This produces the following result −

Output

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

 Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semoi colon (;) at the end.

Following is an example of the abstract method.

Example

```
public abstract class Employee {
   private String name;
   private String address;
   private int number;

   public abstract double computePay();
   // Remainder of class definition
}
```

Declaring a method as abstract has two consequences −

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note − Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below −

```
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary;   // Annual salary

   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
   // Remainder of class definition
}
```

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java −

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example

Following is an example that demonstrates how to achieve Encapsulation in Java −

```
/* File name : EncapTest.java */
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

   public void setAge( int newAge) {
      age = newAge;
   }
   public void setName(String newName) {
      name = newName;
   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
```

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program −

```
/* File name : RunEncap.java */
public class RunEncap {

   public static void main(String args[]) {
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName() + " Age : " + 
encap.getAge());
   }
}
```

This will produce the following result −

Output

```
Name : James Age : 20
```

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields

Java - Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

Following is an example of an interface −

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements

public interface NameOfInterface {
   // Any number of final, static fields
   // Any number of abstract method declarations\
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void travel();
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

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.

Example

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats");
   }

   public void travel() {
      System.out.println("Mammal travels");
   }

   public int noOfLegs() {
      return 0;
   }

   public static void main(String args[]) {
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

This will produce the following result −

Output

```
Mammal eats
Mammal travels
```

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

- 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 implementation 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.

Extending Interfaces

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.

The following Sports interface is extended by Hockey and Football interfaces.

Example

```
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

Example

```
package java.util;
public interface EventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

Creates a common parent − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism

Up Casting and Down Casting in Java:

Typecasting is converting one data type to another.

Up-casting: Converting a subclass type to a superclass type is known as up casting.

Example:

```
class Super {
   void Sample() {
      System.out.println("method of super class");
   }
}

public class Sub extends Super {
   void Sample() {
      System.out.println("method of sub class");
   }

   public static void main(String args[]) {
      Super obj =(Super) new Sub();
obj.Sample();
   }
}
```

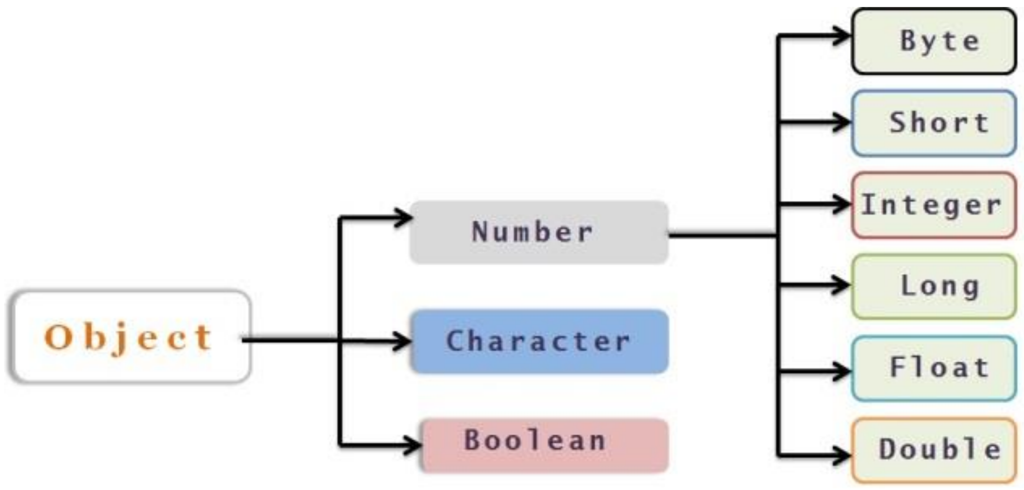
Down-casting: Converting a superclass type to a subclass type is known as downcasting.

```
class Super {
   void Sample() {
      System.out.println("method of super class");
   }
}

public class Sub extends Super {
   void Sample() {
      System.out.println("method of sub class");
   }

   public static void main(String args[]) {
      Super obj = new Sub();
      Sub sub = (Sub) obj;
        sub.Sample();
   }
}

Example:
public class Animal {
    public void walk()
    {
        System.out.println("Walking Animal");
    }
}
class Dog extends Animal {
    public void walk()
    {
        System.out.println("Walking Dog");
    }
    public void sleep()
    {
        System.out.println("Sleeping Dog");
    }
}
```

```
class Demo {
    public static void main (String [] args) {
        Animal a = new Animal();
        Dog d = new Dog();
        a.walk();
        d.walk();
        d.sleep();

        //upcasting
        Animal a2 = (Animal)d;
        a2.walk();
        //a2.sleep();  error

        //downcasting
        Animal a3 = new Dog();
        //Dog d2 = a3;  //compile time error
        Dog d2 = (Dog)a3;
        d2.walk();
        d2.sleep();

        //Run time error: Animal cannot be cast to Dog
        Animal a4 = new Animal();
        //Dog d3 = (Dog)a4;
        //d3.walk();
        //d3.sleep();
    }
}
```

```
Walking Animal
Walking Dog
Sleeping Dog
Walking Dog
Walking Dog
Sleeping Dog
```

Wrapper class Hierarchy



- int age=20; // defining "age" as primitive type

- Integer id = new Integer(30);

We can convert wrapper object to primitive and primitive to wrapper object.

`The process of converting primitive to wrapper is called as " Boxing ".`
`The process of converting wrapper to primitive is called as " Un-boxing "`

Since **boxing** and **un-boxing** features **converts primitive into object** and **object into primitive** automatically and hence its named as **Auto boxing** and **Auto un-boxing**.

Example for Boxing

```
public class WrapperBoxing{
public static void main(String args[]){
//Converting int into Integer
int age=25;
Integer ageObj=Integer.valueOf(age);//converting int into Integer explicitly - Boxing
Integer ageObj1=age;//autoboxing, now compiler will add Integer.valueOf(age) automatically  - Auto boxing

System.out.println(age+" "+ageObj+" "+ageObj1);
}
}
```

Output

```
25 25 25
```

Example for Unboxing

```
public class WrapperUnboxing {
public static void main(String args[]){
//Converting Integer to int
Integer ageObj=new Integer(25);
int age1=ageObj.intValue();//converting Integer to int  explicitly - unboxing
int age2=ageObj;//unboxing, now compiler will add  ageObj.intValue() automatically     - Auto unboxing

System.out.println(ageObj+" "+age1+" "+age2);
}
}
```

Output
25 25 25

Java - Documentation Comments

The Java language supports three types of comments −

Sr.No.	Comment & Description
1	**/* text */** The compiler ignores everything from /* to */.
2	**//text** The compiler ignores everything from // to the end of the line.
3	**/** documentation */** This is a documentation comment and in general its called **doc comment**. The **JDK javadoc** tool uses *doc comments* when preparing automatically generated documentation.

This chapter is all about explaining Javadoc. We will see how we can make use of Javadoc to generate useful documentation for Java code.

What is Javadoc?

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code, which requires documentation in a predefined format.

Following is a simple example where the lines inside /*….*/ are Java multi-line comments. Similarly, the line which preceeds // is Java single-line comment.

Example

```
/**
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld {

   public static void main(String[] args) {
      // Prints Hello, World! on standard output.
      System.out.println("Hello World!");
   }
}
```

You can include required HTML tags inside the description part. For instance, the following example makes use of <h1>....</h1> for heading and <p> has been used for creating paragraph break −

Example

```
/**
* <h1>Hello, World!</h1>
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
* <p>
* Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld {

   public static void main(String[] args) {
      // Prints Hello, World! on standard output.
      System.out.println("Hello World!");
   }
}
```

The javadoc Tags

The javadoc tool recognizes the following tags −

Tag	Description	Syntax
@author	Adds the author of a class.	@author name-text
{@code}	Displays text in code font without interpreting the text as HTML markup or nested javadoc tags.	{@code text}
{@docRoot}	Represents the relative path to the generated document's root directory from any generated page.	{@docRoot}
@deprecated	Adds a comment indicating that this API should no longer be used.	@deprecated deprecatedtext
@exception	Adds a **Throws** subheading to the generated documentation, with the classname and description text.	@exception class-name description
{@inheritDoc}	Inherits a comment from the **nearest** inheritable class or implementable interface.	Inherits a comment from the immediate surperclass.
{@link}	Inserts an in-line link with the visible text label that points to the documentation for the specified	{@link package.class#member label}

	package, class, or member name of a referenced class.	
{@linkplain}	Identical to {@link}, except the link's label is displayed in plain text than code font.	{@linkplain package.class#member label}
@param	Adds a parameter with the specified parameter-name followed by the specified description to the "Parameters" section.	@param parameter-name description
@return	Adds a "Returns" section with the description text.	@return description
@see	Adds a "See Also" heading with a link or text entry that points to reference.	@see reference
@serial	Used in the doc comment for a default serializable field.	@serial field-description | include | exclude
@serialData	Documents the data written by the writeObject( ) or writeExternal( ) methods.	@serialData data-description
@serialField	Documents an ObjectStreamField component.	@serialField field-name field-type field-description
@since	Adds a "Since" heading with the specified since-text to the generated documentation.	@since release
@throws	The @throws and @exception tags are synonyms.	@throws class-name description
{@value}	When {@value} is used in the doc comment of a static field, it displays the value of that constant.	{@value package.class#field}
@version	Adds a "Version" subheading with the specified version-text to the generated docs when the -version option is used.	@version version-text

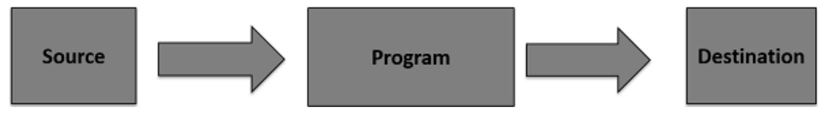
Java - Files and I/O

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one –



Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

Example

Live Demo
```
import java.io.*;
public class ReadConsole {

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

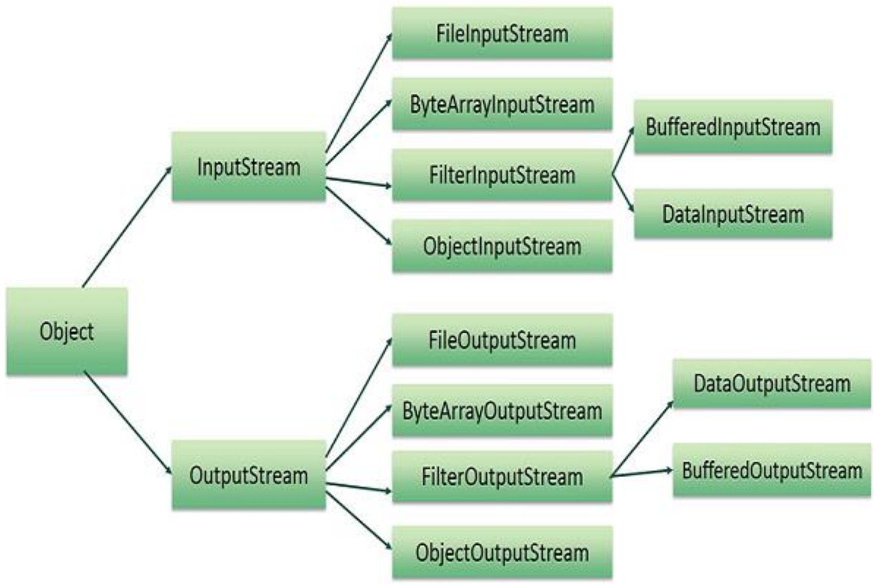
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



Java - Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions** − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

For example, if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

Live Demo

```
import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }
}
```

If you try to compile the above program, you will get the following exceptions.

Output

```
C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8:  error:  unreported  exception  FileNotFoundException;
must be caught or declared to be thrown
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Note − Since the methods **read()** and **close()** of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

- **Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsExceptionexception* occurs.

Example

Live Demo

```
public class Unchecked_Demo {
   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

If you compile and execute the above program, you will get the following exception.

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Exceptions.Unchecked_Demo.main(Unchecked_Demo.java:8)
```

- **Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.

Following is a list of most common checked and unchecked Java's Built-in Exceptions.

Exceptions Methods

Following is the list of important methods available in the Throwable class.

Sr.No.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()**

	Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

Syntax

```
try {
   // Protected code
} catch (ExceptionName e1) {
   // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

Live Demo

```
// File Name : ExcepTest.java
import java.io.*;

public class ExcepTest {

   public static void main(String args[]) {
      try {
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try {
   file = new FileInputStream(fileName);
   x = (byte) file.read();
} catch (IOException i) {
   i.printStackTrace();
   return -1;
} catch (FileNotFoundException f) // Not valid! {
   f.printStackTrace();
   return -1;
}
```

Catching Multiple Type of Exceptions

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code. Here is how you would do it −

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
   throw ex;
```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException −

Example

```
import java.io.*;
public class className {

   public void deposit(double amount) throws RemoteException {
      // Method implementation
      throw new RemoteException();
   }
   // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException −

Example

```
import java.io.*;
public class className {

   public void withdraw(double amount) throws RemoteException,
      InsufficientFundsException {
      // Method implementation
   }
   // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

Syntax

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}finally {
   // The finally block always executes.
}
```

Example

Live Demo

```
public class ExcepTest {

   public static void main(String args[]) {
      int a[] = new int[2];
      try {
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }finally {
         a[0] = 6;
         System.out.println("First element value: " + a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

This will produce the following result −

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following −

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

The try-with-resources

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

Example

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadData_Demo {

   public static void main(String args[]) {
      FileReader fr = null;
      try {
         File file = new File("file.txt");
         fr = new FileReader(file); char [] a = new char[50];
         fr.read(a);   // reads the content to the array
         for(char c : a)
         System.out.print(c);   // prints the characters one by one
      } catch (IOException e) {
         e.printStackTrace();
      }finally {
         try {
            fr.close();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

try-with-resources, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

Syntax

```
try(FileReader fr = new FileReader("file path")) {
   // use the resource
   } catch () {
      // body of catch
   }
}
```

Following is the program that reads the data in a file using try-with-resources statement.

Example

```
import java.io.FileReader;
import java.io.IOException;

public class Try_withDemo {

   public static void main(String args[]) {
      try(FileReader fr = new FileReader("E://file.txt")) {
         char [] a = new char[50];
         fr.read(a);   // reads the contentto the array
         for(char c : a)
         System.out.print(c);   // prints the characters one by one
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Following points are to be kept in mind while working with try-with-resources statement.

- To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
- You can declare more than one class in try-with-resources statement.
- While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block.
- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

User-defined Exceptions

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes −

- All exceptions must be a child of Throwable.

- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below −

```
class MyException extends Exception {
}
```

You just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception {
   private double amount;

   public InsufficientFundsException(double amount) {
      this.amount = amount;
   }

   public double getAmount() {
      return amount;
   }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount {
   private double balance;
   private int number;

   public CheckingAccount(int number) {
      this.number = number;
   }

   public void deposit(double amount) {
      balance += amount;
   }

   public void withdraw(double amount) throws InsufficientFundsException {
      if(amount <= balance) {
         balance -= amount;
      }else {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }

   public double getBalance() {
      return balance;
   }

   public int getNumber() {
      return number;
   }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo {

   public static void main(String [] args) {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);

      try {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      } catch (InsufficientFundsException e) {
         System.out.println("Sorry, but you are short $" + e.getAmount());
         e.printStackTrace();
      }
   }
}
```

Compile all the above three files and run BankDemo. This will produce the following result −

Output

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
         at CheckingAccount.withdraw(CheckingAccount.java:25)
         at BankDemo.main(BankDemo.java:13)
```

Common Exceptions

In Java, it is possible to define two catergories of Exceptions and Errors.

- **JVM Exceptions** − These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.
- **Programmatic Exceptions** − These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

Static blocks in Java

Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. This code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class). For example, check output of following Java program.

```
// filename: Main.java
class Test {
    static int i;
    int j;

    // start of static block
    static {
        i = 10;
        System.out.println("static block called ");
    }
    // end of static block
}

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

        // Although we don't have an object of Test, static block is
        // called because i is being accessed in following statement.
        System.out.println(Test.i);
    }
}
```

Output:

```
static block called
10
```

Also, static blocks are executed before constructors. For example, check output of following Java program.

filter_none

edit

play_arrow

brightness_4

```
// filename: Main.java
class Test {
    static int i;
    int j;
    static {
        i = 10;
        System.out.println("static block called ");
    }
    Test(){
        System.out.println("Constructor called");
    }
}

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

       // Although we have two objects, static block is executed only once.
       Test t1 = new Test();
       Test t2 = new Test();
    }
}
```

Output:

```
static block called
Constructor called
Constructor called
```

Java - Inner classes

Nested Classes

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the **nested class**, and the class that holds the inner class is called the **outer class**.

Syntax

Following is the syntax to write a nested class. Here, the class **Outer_Demo** is the outer class and the class **Inner_Demo** is the nested class.

```
class Outer_Demo {
   class Inner_Demo {
   }
}
```

Nested classes are divided into two types −

- **Non-static nested classes** − These are the non-static members of a class.
- **Static nested classes** − These are the static members of a class.

Inner Classes (Non-static Nested Classes)

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are −

- Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Inner Class

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

Example

Live Demo

```
class Outer_Demo {
   int num;
   
   // inner class
   private class Inner_Demo {
      public void print() {
         System.out.println("This is an inner class");
      }
   }
   
   // Accessing he inner class from the method within
   void display_Inner() {
      Inner_Demo inner = new Inner_Demo();
      inner.print();
   }
}

public class My_class {

   public static void main(String args[]) {
      // Instantiating the outer class
      Outer_Demo outer = new Outer_Demo();

      // Accessing the display_Inner() method.
      outer.display_Inner();
   }
}
```

Here you can observe that **Outer_Demo** is the outer class, **Inner_Demo** is the inner class, **display_Inner()** is the method inside which we are instantiating the inner class, and this method is invoked from the **main** method.

If you compile and execute the above program, you will get the following result −

Output

```
This is an inner class.
```

Accessing the Private Members

As mentioned earlier, inner classes are also used to access the private members of a class. Suppose, a class is having private members to access them. Write an inner class in it, return the private members from a method within the inner class, say, **getValue()**, and finally from another class (from which you want to access the private members) call the getValue() method of the inner class.

To instantiate the inner class, initially you have to instantiate the outer class. Thereafter, using the object of the outer class, following is the way in which you can instantiate the inner class.

```
Outer_Demo outer = new Outer_Demo();
Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
```

The following program shows how to access the private members of a class using inner class.

Example

Live Demo

```
class Outer_Demo {
   // private variable of the outer class
   private int num = 175;

   // inner class
   public class Inner_Demo {
      public int getNum() {
         System.out.println("This is the getnum method of the inner class");
         return num;
      }
   }
}

public class My_class2 {

   public static void main(String args[]) {
      // Instantiating the outer class
      Outer_Demo outer = new Outer_Demo();

      // Instantiating the inner class
      Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
      System.out.println(inner.getNum());
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is the getnum method of the inner class: 175
```

Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

Example

Live Demo

```
public class Outerclass {
   // instance method of the outer class
   void my_Method() {
      int num = 23;

      // method-local inner class
      class MethodInner_Demo {
         public void print() {
            System.out.println("This is method inner class "+num);
         }
      } // end of inner class

      // Accessing the inner class
      MethodInner_Demo inner = new MethodInner_Demo();
      inner.print();
   }

   public static void main(String args[]) {
      Outerclass outer = new Outerclass();
      outer.my_Method();
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is method inner class 23
```

Anonymous Inner Class

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows −

Syntax

```
AnonymousInner an_inner = new AnonymousInner() {
   public void my_method() {
      ........
      ........
   }
};
```

The following program shows how to override the method of a class using anonymous inner class.

Example

Live Demo

```
abstract class AnonymousInner {
   public abstract void mymethod();
}

public class Outer_class {

   public static void main(String args[]) {
      AnonymousInner inner = new AnonymousInner() {
         public void mymethod() {
            System.out.println("This is an example of anonymous inner class");
         }
      };
      inner.mymethod();
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is an example of anonymous inner class
```

In the same way, you can override the methods of the concrete class as well as the interface using an anonymous inner class.

Anonymous Inner Class as Argument

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument −

```
obj.my_Method(new My_Class() {
   public void Do() {
      .....
      .....
   }
});
```

The following program shows how to pass an anonymous inner class as a method argument.

Example

Live Demo

```
// interface
interface Message {
   String greet();
}

public class My_class {
   // method which accepts the object of interface Message
   public void displayMessage(Message m) {
      System.out.println(m.greet() +
         ", This is an example of anonymous inner class as an argument");
   }

   public static void main(String args[]) {
      // Instantiating the class
      My_class obj = new My_class();

      // Passing an anonymous inner class as an argument
      obj.displayMessage(new Message() {
         public String greet() {
            return "Hello";
         }
      });
   }
}
```

If you compile and execute the above program, it gives you the following result −

Output

```
Hello, This is an example of anonymous inner class as an argument
```

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows −

Syntax

```
class MyOuter {
   static class Nested_Demo {
   }
}
```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

Example

Live Demo

```
public class Outer {
   static class Nested_Demo {
      public void my_method() {
         System.out.println("This is my nested class");
      }
   }

   public static void main(String args[]) {
      Outer.Nested_Demo nested = new Outer.Nested_Demo();
      nested.my_method();
   }
}
```

If you compile and execute the above program, you will get the following result −

Output

```
This is my nested class
```

J2EE

let’s understand about different editions of Java platform-

- **J2SE(Java Platform, Standard Edition)**

Also known as Core Java, this is the most basic and standard version of Java.It’s the purest form of Java, a basic foundation for all other editions.

It consists of a wide variety of general purpose API’s (like java.lang, java.util) as well as many special purpose APIs

J2SE is mainly used to create applications for Desktop environment.

It consist all the basics of Java the language, variables, primitive data types, Arrays, Streams, Strings Java Database Connectivity(JDBC) and much more. This is the standard, from which all other editions came out, according to the needs of the time.

The famous JVM of Java, the heart of Java development, was also given by this edition only.It’s because of this feature, that Java has such a wide usage.

- **J2ME(Java Platform, Micro Edition)**

This version of Java is mainly concentrated for the applications running on embedded systems, mobiles and small devices.(which was a constraint before it’s development)

Constraints included limited processing power, battery limitation, small display etc.

Also, the J2ME apps help in using web compression technologies, which in turn, reduce network usage, and hence cheap internet accessibility.

J2ME uses many libraries and API’s of J2SE, as well as, many of it’s own.

The basic aim of this edition was to work on mobiles, wireless devices, set top boxes etc.

Old Nokia phones, which used Symbian OS, used this technology.

Most of the apps, developed for the phones(prior to smartphones era), were built on J2ME platform only(the .jar apps on Nokia app store).

- **J2EE(Java Platform, Enterprise Edition)**

The Enterprise version of Java has a much larger usage of Java, like development of web services, networking, server side scripting and other various web based applications.

J2EE is a community driven edition, i.e. there is a lot of continuous contributions from industry experts, Java developers and other open source organizations.

J2EE uses many components of J2SE, as well as, has many new features of it's own like Servlets, JavaBeans, Java Message Services, adding a whole new functionalities to the language.

J2EE uses HTML, CSS, JavaScript etc., so as to create web pages and web services. It's also one of the most widely accepted web development standard.

There are also many languages like .net and php, which can do that work, but what distinguishes it from other languages is the versatility, compatibility and security features, which are not that much prominent in other languages.

Nowadays, developers are going more towards this edition, as it more versatile and web friendly that it's other counterparts.

Apart from these three versions, there was another Java version, released **Java Card**.

This edition was targeted, to run applets smoothly and securely on smart cards and similar technology.

Portability and security was its main features.

JavaFX is another such edition of Java technology, which is now merged with J2SE 8.It is mainly used, to create rich GUI (Graphical User Interface) in Java apps.

It replaces Swings (in J2SE), with itself as the standard GUI library.

It is supported by both Desktop environment as well as web browsers.

PersonalJava was another edition, which was not deployed much, as its function was fulfilled by further versions of J2ME. Made to support World Wide Web (and Java applets) and consumer electronics.

PersonalJava was also used for embedded systems and mobile. But, it was discontinued in its earlier stages.

Basic Introduction to Java 2 Enterprise Edition (J2EE)

J2EE was proposed by Sun Microsystems (Now, Oracle) for developing and deploying multitier, distributed, enterprise scale business applications. Applications written for J2EE standards enjoy certain benefits such as portability, security, scalability, load-balancing, and reusability. Enterprise Edition (J2EE) builds on top of Java2 Standard Edition (J2SE) to provide the types of services that are necessary to build large scale, distributed, component based, multi-tier applications. In other words, J2EE is a collection of APIs which could be used to build such large-scale systems.

J2EE is aimed to be a standard for building and deploying enterprise applications, held together by the specifications of the APIs that it defines and the services that J2EE provides. In other words, this means that the “write once, run anywhere” promises of Java apply for enterprise applications too:

- Enterprise applications can be run on different platforms supporting the Java 2 platform.
- Enterprise applications are portable between application servers supporting the J2EE specification.

What does J2EE comprise?

J2EE is composed of a number of APIs that can be used to build enterprise applications. Although the total list of APIs initially seems overwhelming, it is worth bearing in mind that some are primarily used by the J2EE environment in which your application executes, while some provide services that your specific application may not require. Therefore, it is worth remembering that you don’t have to use all of them in order to build J2EE applications. For completeness, however, the full list of technologies that make up J2EE is as follows:

- Java Servlets
- JavaServer Pages (JSP)
- Enterprise JavaBeans (EJB)
- Java Message Service (JMS)
- Java Naming and Directory Interface (JNDI)
- Java Database Connectivity (JDBC)
- JavaMail
- Java Transaction Service (JTS)
- Java Transaction API (JTA)
- J2EE Connector Architecture (J2EE-CA, or JCA)

From a developer perspective, the main technologies are EJB, JSP, Java Servlets, JDBC and JMS, although JNDI is used for locating EJBs and other enterprise resources.

Java Servlets

Java Servlets are the Java equivalent of PHP or ASP that can be used to perform processing and the servicing of client requests on a web server. From an implementation perspective, servlets are simply Java classes that implement a predefined interface. One use for servlets is that they can be used to dynamically generate content for presentation to the user, and this is achieved by embedding markup language (e.g. HTML) inside the Java code. As Servlets are written in Java, they have access to the rich library of features provided by Java, including access to databases and other enterprise resources such as EJB.

JavaServer Pages (JSP)

JSP is another technology for presenting information to the user over the web and uses a paradigm where Java code is embedded into the HTML – the opposite of servlets, and much like Microsoft ASP. Pages are written as HTML files with embedded Java source code known as scriptlets.

Enterprise JavaBeans (EJB)

EJB is a major part of the J2EE specification and defines a model for building server-side, reusable components.

Java Message Service (JMS)

JMS is Java API that presents an interface into message-oriented middleware such as IBM MQSeries, SonicMQ and so on. Like JDBC, JMS provides Java applications a mechanism to integrate with such systems by presenting a common programming interface irrespective of the underlying messaging system. Functionally, JMS allows messages to be sent and received using a point-to-point or publish/subscribe paradigm.

Why J2EE?

You'll want to use the J2EE platform for developing your Java e-commerce or Enterprise application if any of these statements apply:

- You want to make use of a standardized, industry-tested framework that provides support for transaction management, naming, security, remote connectivity, and database access.
- You require portability-the ability for your components to be deployed on your choice of J2EE-compliant server.
- You want to reuse your components or use purchased off-the-shelf components.
- You want to make use of your developers' architectural experience on the J2EE platform, using tried and true architecture and design patterns.
- Your system needs to be scalable to meet increased loads.
- You want to reduce development time by using powerful J2EE development and deployment tools.
- You want to easily integrate with back end systems.
- You want to take advantage of simple, powerful security features.

Each of these points is discussed in further detail in the rest of this section.

Standardized, Industry-Tested

J2EE components run in J2EE containers, typically provided as part of a J2EE-compliant server. These containers provide a set of standard services (APIs) used by the J2EE components. The APIs are:

- J2SE 1.4
  - JDBC
  - Java IDL
  - Remote Method Invocation with CORBA's Internet Inter-ORB Protocol (RMI-IIOP)
  - Java Naming and Directory Interface (JNDI)
  - Java Authentication and Authorization Service (JAAS)
- Java Transaction API (JTA)
- JavaMail
- Java Message Service (JMS). For more information on JMS, see Concept: Java Messaging Service (JMS).
- JavaBeans Activation Framework (JAF)
- Enterprise JavaBeans (EJB)
- Java Servlet
- Java API for XML Processing (JAXP)
- Java Connector (**Note**: not supported prior to J2EE 1.3)
- Java Server Pages (JSP)
- Web Services for J2EE (**Note**: not supported prior to J2EE 1.4)
- Java API for XML-based RPC (JAX-RPC) (**Note**: not supported prior to J2EE 1.4)
- SOAP with attachments API for Java (SAAJ) (**Note**: not supported prior to J2EE 1.4)
- Java API for XML Registries (JAXR) (**Note**: not supported prior to J2EE 1.4)
- J2EE Management (**Note**: not supported prior to J2EE 1.4)
- Java Management Extensions (JMX) (**Note**: not supported prior to J2EE 1.4)
- J2EE Deployment (**Note**: not supported prior to J2EE 1.4)
- Java Authorization Service Provider Contract for Containers (JACC) (**Note**: not supported prior to J2EE 1.4)

Portability

J2EE components and applications are portable across J2EE-compliant servers, with no code modifications necessary, so you can deploy your application to the J2EE-compliant server of your choice simply by updating server-specific deployment information contained in eXtended Markup Language (XML) deployment descriptor files.

The standardization of the J2EE specification has led to industry competition-you have a choice of J2EE-compliant servers according to your needs and budget.

Reusable Components

Because they conform to the J2EE standard, J2EE components can be bought off-the-shelf and plugged into your J2EE application as required, saving development (especially debugging and testing) effort.

If you develop a component, you can reuse it in another application or deploy it to different J2EE-compliant servers, as required.

Tried and True Architecture and Design Patterns

The J2EE platform defines a well-structured, multi-tiered application architecture. By leveraging off the J2EE architecture, your developers can quickly get on with developing the actual business logic of the application.

J2EE documentation includes:

- A blueprint for application development that describes the J2EE platform in detail and gives best-practice information on how to develop J2EE applications.
- Well-documented J2EE patterns-industry best practices-that describe solutions to common J2EE architectural and design problems.

For more information on the J2EE Platform, see http://java.sun.com/. Follow the links to **J2EE > Blueprints**.

Scalability

J2EE supports scalability to increase performance or to meet increased loads in several ways:

- **Performance enhancement features in the J2EE container** - such features include resource pooling (database connection pooling, session bean instance pooling, and thread pooling), asynchronous message passing, and efficient component lifecycle management. For example, opening a database connection is slow. Also, database connections could be a scarce resource due to, for instance, licensing restrictions. The J2EE platform manages this using database connection pooling; the J2EE container keeps a pool of open connections that can be assigned to a component as required, resulting in fast and efficient connections.
- **Load balancing can be achieved by clustering** - deploying the same components to multiple servers on different machines. The load to each of the servers can then be balanced as required; for example, according to a round-robin algorithm or according to server load. The J2EE platform specification does not require load balancing capability in a J2EE server, but does suggest that a high-end server would have it. J2EE server vendors offer various load-balancing solutions.
- **Application partitioning** - logically distinct parts of an application can be deployed to different servers; for example, deploying an online mail order application's inventory and accounting subsystems to separate servers.

Development and Deployment Tools

Vendors have responded to the need for J2EE tools by providing excellent support for J2EE development in their Java Integrated Development Environments (IDEs) including:

- Wizards for servlet creation
- Wizards and dialogs for EJB creation and maintenance
- Deployment descriptor generation and maintenance
- EJB object to database mapping (including generation of deployment descriptor information for container-managed relationships)
- Integration with a Web container for testing Web services
- Seamless in-IDE deployment, debug, and testing of EJBs by integration with a J2EE EJB container and its deployment tools
- Automatic generation of J2EE test clients
- Integration with UML modeling tools

Back End Integration

The back end refers to the enterprise information system (EIS) tier of the application. Back end systems can be, for example, RDBMS, legacy systems, or enterprise resource planning systems (ERPs).

J2EE supports transactional access to RDBMS EISs using the JDBC and JTA APIs. In addition, EJB containers support container-managed persistence, in which transactional RDBMS connection and access is handled automatically by the container.

J2EE's Connector Architecture Service Provider Interface (SPI) defines a standard for connecting non-RDBMS EIS resources to a J2EE container. An EIS-specific resource adapter (supplied by the EIS vendor) is plugged in to the J2EE container, extending the container so that it provides transactional, secure support for that EIS. Components in the container can then access the EIS through the J2EE Connector Architecture SPI.

Note: J2EE's Connector Architecture SPI is not supported prior to J2EE 1.3.

Security

J2EE provides simple, powerful security features. Security information for J2EE components is defined in their deployment descriptors. This information defines what security *roles* are authorized to access a particular URL and/or methods of a component. A security role is merely a logical name for a grouping of users; for example, an organization's management team members could all be assigned a role named "managers".

Since the security information is declared in the deployment descriptor, the security behavior can be changed without an expensive code update-debug-test cycle.

Multi-Tier Architecture

J2EE is a multi-tier distributed application architecture-consisting of a client tier, middle tier, and EIS or back end tier.

Figure 1 shows the multi-tier architecture of the J2EE platform, as well as the various J2EE containers supporting J2EE components.

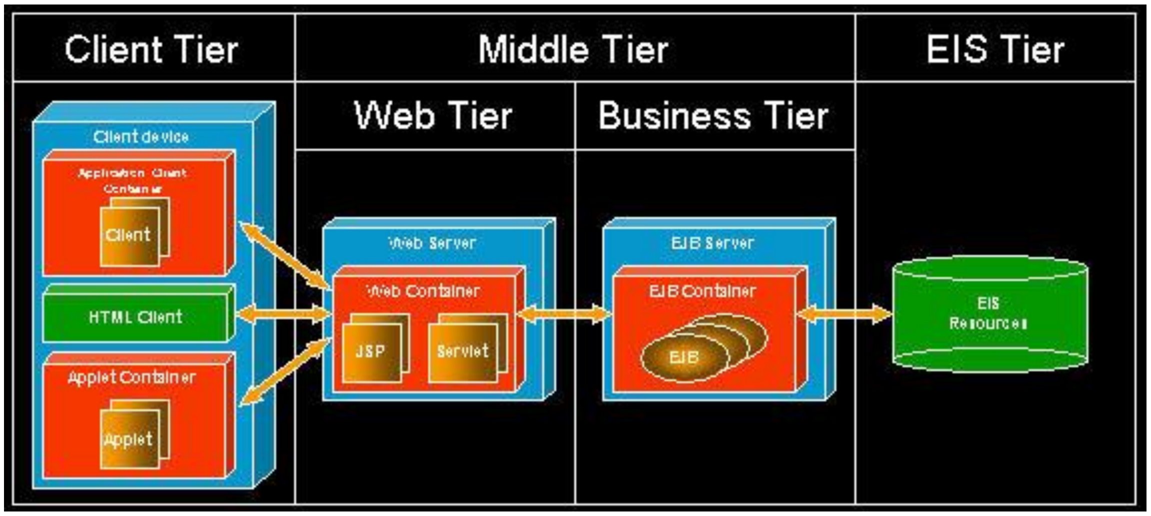


Figure 1: J2EE Multi-tier Architecture

Client Tier

Client tier components run in client containers. The client tier can be implemented in these ways:

- Standalone Java applications - usually a GUI (also known as a "thick client"). Such a Java application must be installed on every client machine. A Java application can access the EIS tier or middle tier through APIs such as JDBC.
- Static HTML pages - provide a limited GUI for an application.
- Dynamic HTML - generated by JSP pages or servlets.
- Applets - run in a Web browser. Applets are embedded in an HTML page and are typically used to provide a GUI.

Middle Tier

The middle tier consists of the Web tier and business tier. Web tier components run in a J2EE Web server that provides a Web container. Business tier components run in a J2EE application server that provides an EJB container.

Web Tier

Web tier components include servlets and JSP pages, which manage the interaction with the client tier, insulating the clients from the business and EIS tier. Clients make requests of the Web tier, which processes the requests and returns the results to the client. Client requests to components in the Web tier generally result in Web tier requests to components in the business tier, which, in turn, might result in requests to the EIS tier.

Business Tier

Business tier components are EJBs.

- They contain the application business logic.
- They make requests to the EIS tier according to the business logic, typically in response to a request from the Web tier.

EIS Tier

The EIS tier represents the application's stored data, often in the form of an RDBMS. The EIS tier might also consist of legacy systems or ERPs, accessed through the J2EE Connector Architecture API.

For more information on the J2EE Connector Architecture API, see http://java.sun.com/. Follow the links to Products & Technologies > J2EE > J2EE Connector Architecture.

For more information on J2EE's standard deployment configurations, see Concept: J2EE Deployment Configurations.

J2EE Servers

J2EE servers are commercial products that implement the J2EE platform. Examples of commercial J2EE servers are BEA WebLogic, Borland Enterprise Server, IBM WebSphere, and iPlanet.

Usage of the terminology "J2EE server" is somewhat loose. Usually, what's meant is "a J2EE server that supports both a Web container and an EJB container". Using tighter terminology, a J2EE Web server (such as the J2EE reference Web server implementation Tomcat) supports a Web container; a J2EE application (or EJB) server supports an EJB container.

J2EE Containers

J2EE components run in, or are hosted by, J2EE containers generally provided as part of a commercial J2EE server. Containers provide a run-time environment and standard set of services (APIs) to the J2EE components running in the container, in addition to supporting the standard J2SE APIs.

J2EE defines the following types of containers:

- application client container
- applet container
- Web container
- EJB container

N Tier(Multi-Tier), 3-Tier, 2-Tier Architecture with EXAMPLE

What is N-Tier?

An **N-Tier Application** program is one that is distributed among three or more separate computers in a distributed network.

The most common form of n-tier is the 3-tier Application, and it is classified into three categories.

- User interface programming in the user's computer
- Business logic in a more centralized computer, and
- Required data in a computer that manages a database.

This architecture model provides Software Developers to create Reusable application/systems with maximum flexibility.

In **N-tier, "N"** refers to a number of tiers or layers are being used like – **2-tier, 3-tier or 4-tier, etc**. It is also called "**Multi-Tier Architecture"**.

The n-tier architecture is an industry-proven software architecture model. It is suitable to support enterprise level client-server applications by providing solutions to scalability, security, fault tolerance, reusability, and maintainability. It helps developers to create flexible and reusable applications.

In this tutorial, you will learn-

N-Tier Architecture

A diagrammatic representation of an n-tier system depicts here – presentation, application, and database layers.

N Tier Architecture Diagram

These three layers can be further subdivided into different sub-layers depending on the requirements.

Some of the popular sites who have applied this architecture are

- MakeMyTrip.com
- Sales Force enterprise application
- Indian Railways – IRCTC
- Amazon.com, etc.

Some common terms to remember, so as to understand the concept more clearly.

- **Distributed Network:** It is a network architecture, where the components located at network computers coordinate and communicate their actions only by passing messages. It is a collection of multiple systems situated at different nodes but appears to the user as a single system.
  - It provides a single data communication network which can be managed separately by different networks.
  - An example of Distributed Network– where different clients are connected within LAN architecture on one side and on the other side they are connected to high-speed switches along with a rack of servers containing service nodes.
- **Client-Server Architecture:** It is an architecture model where the client (one program) requests a service from a server (another program) **i.e.** It is a request-response service provided over the internet or through an intranet.

  In this model, **Client** will serve as one set of program/code which executes a set of actions over the network. While **Server**, on the other hand, is a set of another program, which sends the result sets to the client system as requested.

  - In this, client computer provides an interface to an end user to request a service or a resource from a server and on the other hand server then processes the request and displays the result to the end user.
  - An example of Client-Server Model– an ATM machine. A bank is the server for processing the application within the large customer databases and ATM machine is the client having a user interface with some simple application processing.

- **Platform:** In computer science or software industry, a platform is a system on which applications program can run. It consists of a combination of hardware and software that have a built-in instruction for a processors/microprocessors to perform specific operations.
  - In more simple words, the platform is a system or a base where any applications can run and execute to obtain a specific task.
  - An example of Platform – A personal machine loaded with Windows 2000 or Mac OS X as examples of 2 different platforms.

- **Database:** It is a collection of information in an organized way so that it can be easily accessed, managed and updated.
    - Examples of Database – MySQL, SQL Server, and Oracle Database are some common Db's.

Types of N-Tier Architectures

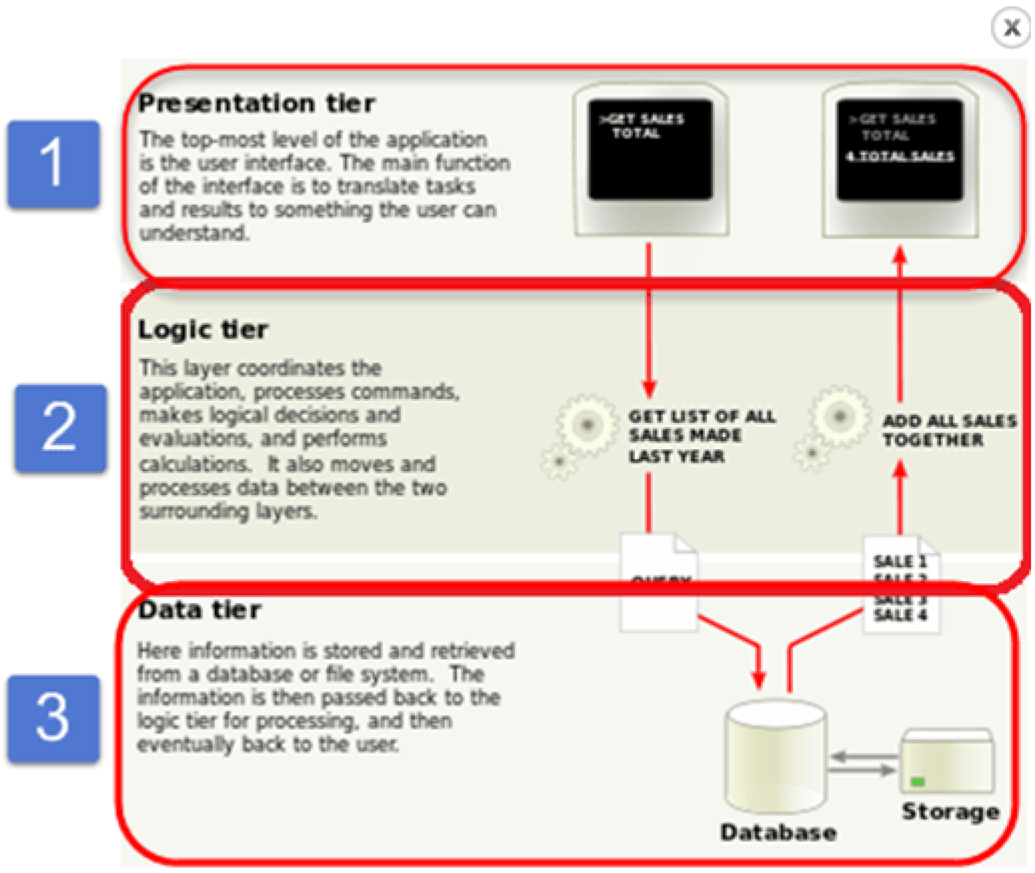
There are different types of N-Tier Architectures, like **3-tier Architecture, 2-Tier Architecture and 1- Tier Architecture.**

First, we will see 3-tier Architecture, which is very important.

3-Tier Architecture

By looking at the below diagram, you can easily identify that **3-tier architecture** has three different layers.

- Presentation layer
- Business Logic layer
- Database layer



The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

3 Tier Architecture Diagram

Here we have taken a simple example of student form to understand all these three layers. It has information about a student like – Name, Address, Email, and Picture.

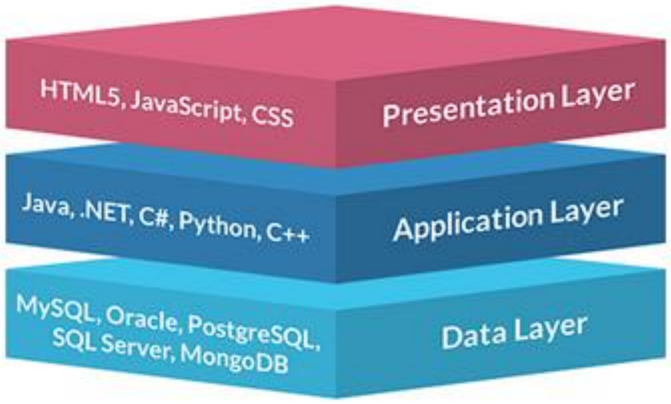
3-Tier Architecture: A Complete Overview

What is a 3-Tier Architecture?

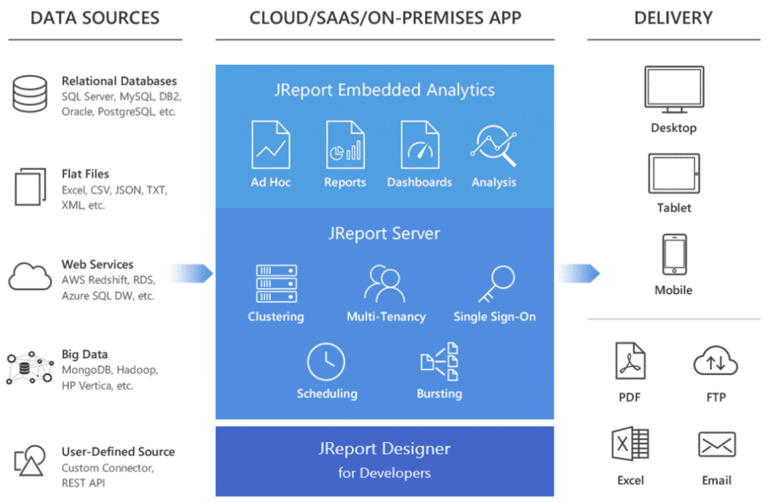
A 3-tier architecture is a type of software architecture which is composed of three “tiers” or “layers” of logical computing. They are often used in applications as a specific type of client-server system. 3-tier architectures provide many benefits for production and development environments by modularizing the user interface, business logic, and data storage layers. Doing so gives greater flexibility to development teams by allowing them to update a specific part of an application independently of the other parts. This added flexibility can improve overall time-to-market and decrease development cycle times by giving development teams the ability to replace or upgrade independent tiers without affecting the other parts of the system.

For example, the user interface of a web application could be redeveloped or modernized without affecting the underlying functional business and data access logic underneath. This architectural system is often ideal for embedding and integrating 3rd party software into an existing application. This integration flexibility also makes it ideal for embedding analytics software into pre-existing applications and is often used by embedded analytics vendors for this reason. 3-tier architectures are often used in cloud or on-premises based applications as well as in software-as-a-service (SaaS) applications.

What Do the 3 Tiers Mean?


HTML5, JavaScript, CSS
Presentation Layer
Java, .NET, C#, Python, C++
Application Layer
MySQL, Oracle, PostgreSQL, SQL Server, MongoDB
Data Layer

- **Presentation Tier-** The presentation tier is the front end layer in the 3-tier system and consists of the user interface. This user interface is often a graphical one accessible through a web browser or web-based application and which displays content and information useful to an end user. This tier is often built on web technologies such as HTML5, JavaScript, CSS, or through other popular web development frameworks, and communicates with others layers through API calls.
- **Application Tier-** The application tier contains the functional business logic which drives an application's core capabilities. It's often written in Java, .NET, C#, Python, C++, etc.
- **Data Tier-** The data tier comprises of the database/data storage system and data access layer. Examples of such systems are MySQL, Oracle, PostgreSQL, Microsoft SQL Server, MongoDB, etc. Data is accessed by the application layer via API calls.



Example of a 3-tier architecture: JReport.

The typical structure for a 3-tier architecture deployment would have the presentation tier deployed to a desktop, laptop, tablet or mobile device either via a web browser or a web-based application utilizing a web server. The underlying application tier is usually hosted on one or more application servers, but can also be hosted in the cloud, or on a dedicated workstation depending on the complexity and processing power needed by the application. And the data layer would normally comprise of one or more relational databases, big data sources, or other types of database systems hosted either on-premises or in the cloud.