



Wolaita Sodo University
Department of Computer Science
Object Oriented Programming & Java Programming
Module

Prepared and Compiled by:

Dawit Uta (MTech.)

Reviewed by:

1. Arba Asha (MSc.), HoD, Department of Computer Science
2. Mesay Wana (MSc.)
3. Melaku Bayih (MSc.)

February, 2023

Contents

Chapter 1: Introduction to Object-Oriented Programming	6
1.1. Types of programming paradigms	6
1.2. Overview of OO principles	7
1.3. Editing, Compiling and Interpreting	7
1.4. Overview of Java Programming:	9
Chapter Two: Classes and Objects in Java	12
2.1. Defining a class	12
2.2. Creating an Object	13
2.3. Instantiating and using objects	15
2.4. Methods (Functions)	18
2.5. Constructors	25
2.6. Garbage Collection	29
Chapter Three: Inheritance and Polymorphism.....	40
3.1. Inheritance.....	40
3.2. Member Access and Inheritance	42
3.3. Using super	46
3.4. Casting	50
3.5. Polymorphism	52
3.6. Method Overriding and Overloading.....	52
3.6.1 Method Overriding.....	52
3.6.2 Overloading Methods.....	58
3.7. Creating a Multilevel Hierarchy	68
3.8. Abstract Classes	73
3.9. The Object Class	76
3.10. Interfaces.....	77
Chapter 4: Exception Handling	79
4.1. Exceptions Overview	79
4.2. Hierarchy of Java Exception classes	79
4.2.1 Types of Java Exceptions.....	79

4.3	Catching Exceptions	80
4.3.1	Java try block	81
4.3.2	Java Multi-catch block	82
4.3.3	Finally Block.....	84
4.3.4	Exceptions Methods	85
4.3.5	Throws Keyword.....	85
4.4	Errors and Runtime Exceptions	87
Chapter 5: Packages		88
5.1	Packages.....	88
5.2	The import Statement.....	88
5.3	Static Imports	89
5.4	Defining Packages	89
5.5	Package class	90
Chapter 6: Data structures.....		91
6.1	The Set	91
6.2	Operations on the Set Interface	92
6.3	Set Implementation Classes	93
6.4	The List	95
6.5	The Queue.....	95
6.6	Map	98
Part Two		100
Java Programming Module		100
Chapter 1: Overview of Java programming		100
1.1	Introduction to Java Programming.....	100
1.2	Types of Java Applications:.....	100
1.3	Java Platforms / Editions	101
1.4	Variables in Java	101
1.5	Identifiers in Java	101
1.6	Data Types in Java	102
1.7	Java Arrays.....	103
1.7.1	Types of Array in java	103

1.7.2 for-each Loop for Java Array	104
1.8 Java Control and Repetition Statements	105
1.8.1 Decision-Making statements:.....	105
1.8.2 Loop Statements.....	109
Chapter 2: Java Applets	116
2.1 Introduction to Java Applets	116
2.2 Life Cycle of an Applet.....	116
2.3 The Applet Class.....	117
2.4 Event Handling in applets	120
Chapter Three: Java GUI using JavaFX	123
3.1 Introduction.....	123
3.3 Features of JavaFX	123
3.4 JavaFX – Architecture	124
3.5 JavaFX Application Structure.....	127
3.6 Creating a JavaFX Application	128
3.7 Lifecycle of JavaFX Application	131
3.8 Terminating the JavaFX Application.....	132
3.9 JAVAFX layout components	132
3.10 JavaFX - UI Controls	134
3.11 JavaFX Event Handling	143
3.11.1Types of Events.....	144
3.11.2 Events in JavaFX	144
3.12 JavaFX Shapes	147
3.12.1 JavaFX Shape Properties	148
3.12.2 JavaFX Line	149
3.12.3 JavaFX Rectangle	152
3.12.4 JavaFX Circle.....	154
3.13 JavaFX CSS	156
3.14 JavaFX Animation	159
3.14.1 Steps for applying Animations.....	160
3.14.2 JavaFX Rotate Transition.....	161

Chapter 4: Streams and File I/O.....	164
4.1 Introduction Streams and File	164
4.2 Standard Streams	166
4.3 Reading and Writing Files	167
4.4 File Management	172
Output	179
Chapter 5: Multi-Threading in Java	182
5.1 Introduction to Multi-threading	182
5.2 Difference between Process and Thread	182
5.3 Life Cycle of a Thread	184
5.4 The Implementation Threads in java	185
5.4.1 Create a Thread by Extending a Thread Class	185
5.4.2 Create a Thread by Implementing a Runnable Interface.....	185
5.5 The Mechanisms for handling Multiple Threads.....	187
5.5.1 Priority of a Thread (Thread Priority)	187
5.5.2 Java - Thread Synchronization.....	188
Chapter 6: Networking in Java.....	193
6.1 Introduction to network programming.....	193
6.2 Socket Programming.....	193
6.2.1 Implementing Socket Programming	194
6.3 Remote Method Invocation.....	200
6.3.1 Architecture of an RMI Application	200
6.3.2 Implementing RMI Application.....	201
6.3.3 Marshalling and Unmarshalling.....	201
6.4 RMI Registry	202
6.4.1 Defining the Remote Interface.....	203
6.4.2 Developing the Implementation Class (Remote Object)	203
6.4.3 Developing the Server Program.....	204
6.4.4 Developing the Client Program.....	205
6.4.5 Compiling the Application.....	205
6.4.6 Executing the Application.....	206

Chapter 7: Java Database Connectivity	208
7.1 Introduction to JDBC.....	208
7.2 Why Should We Use JDBC	209
7.3 Structured Query Language (SQL	209
7.4 Steps of Java Database Connectivity	212
7.4.1 JDBC Driver	212
7.4.2 Establishing Connection	214
7.4.3 DBC - Statements, PreparedStatement and CallableStatement	217
7.4.4 ResultSet	219
7.5 JDBC Sample Code	221
Chapter 8: Servlets.....	225
8.1 Introduction to servlets	225
8.2 The Servlet API.....	225
8.3 Request redirecting	228
8.4 Multi-tier applications using JDBC from servlet	232

Chapter 1: Introduction to Object-Oriented Programming

1.1. Types of programming paradigms

Overview of computer programming

For a computer to be able to do anything (multiply, play a song, run a word processor), it must be given the instructions to do so.

A **program** is a set of instructions written by humans for computers to perform tasks. The instructions are written in **programming languages** such as C, C++, Java, etc.

The term paradigm is can be **pattern**, **model**, or approach to programming that a language supports. It is a pattern that serves as a **school** of **thoughts** for programming of computers. We can characterize a main programming paradigm in terms of an idea and a basic discipline.

Main programming paradigms

i. **Imperative paradigm**: oldest approach, closest to the actual mechanical behavior of a computer⇒ original imperative languages were abstractions of assembly language,

Useful for quite simple programs.

♣ It is also called Structured or Procedural Programming

♣ Languages like FORTRAN, BASIC, COBOL, Pascal, C are good examples

ii. **Functional paradigm**: Nearly as old as imperative programming. Created by John McCarthy with LISP (list processing) in the late 1950s.

- Program as a collection of (math) functions
- Many important innovations that have been deeply influential.
- Main languages: Mathematica, Common Lisp/Scheme/Clojure, Standard,
- ML/Calm/OCalm/F#, Haskell, Erlang/Elixir, Scala.

iii. **Logical paradigm**: fits extremely well when applied in problem domains that deal with the

- Extraction of knowledge from basic facts and relations.
- The logical paradigm seems less natural in the more general areas of computation.
- Based on axioms, inference rules, and queries.
- Program execution becomes a systematic search in a set of facts, making use of a set of inference rules
- **Object-oriented programming (OOP)** is a fundamental and most popular programming paradigm used by nearly every developer at some point in their career. OOP is a programming paradigm that relies on the concept of classes and objects.
- It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.
- Object receive messages, processes data, and sends messages to other objects.
- There are many object-oriented programming languages including C++, Java, JavaScript, Python etc.
- The simplest way to explain object-orientated programming is to use something like a car as an example.
- Incorporates both encapsulation and inheritance through the class concept

- Focus is on writing good classes and on code reuse
- Examples: Shape, Circle, and Rectangle in a drawing program, Employee, Faculty, Staff in a university personnel system
- A car has a model name, a color, a year in which it was manufactured, an engine size and so on.

Benefits of OOP

- OOP models complex things as reproducible, simple structures
- Reusable, OOP objects can be used across programs
- Allows for class-specific behavior through polymorphism
- Easier to debug, classes often contain all applicable information to them
- Secure, protects information through encapsulation × The fundamental building blocks of an OOP program are:
- Classes, Objects, Methods, Attributes

1.2. Overview of OO principles

Principles of OOP: The four pillars of object oriented programming are:

- **Inheritance:** child classes inherit data and behaviors from parent class
- **Encapsulation:** containing information in an object, exposing only selected information
- **Abstraction:** only exposing high level public methods for accessing an object
- **Polymorphism:** many methods can do the same task



1.3. Editing, Compiling and Interpreting

Computers do not understand the languages (C++, Java, etc) that programs are written in. Programs must first be **compiled** (converted) into **machine code** that the computer can run. A **compiler** is a program that translates a programming language into machine code.

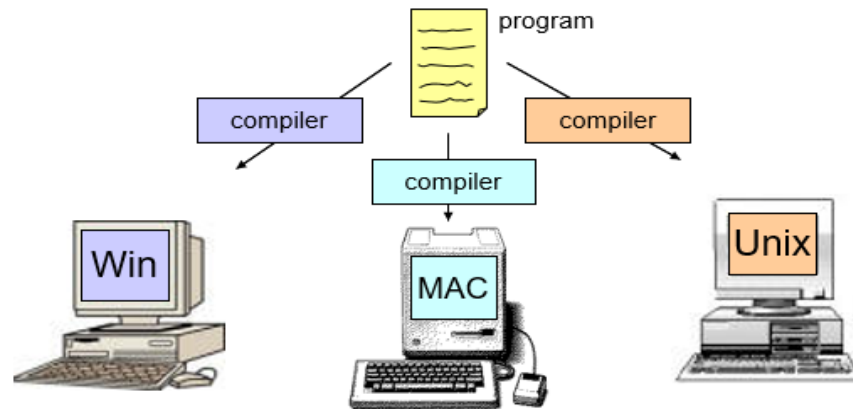
Running Programs All programs follow a simple format:

Inputs can be from users, files, or other computer programs

Outputs can take on many forms: numbers, text, graphics, sound, or commands to other programs

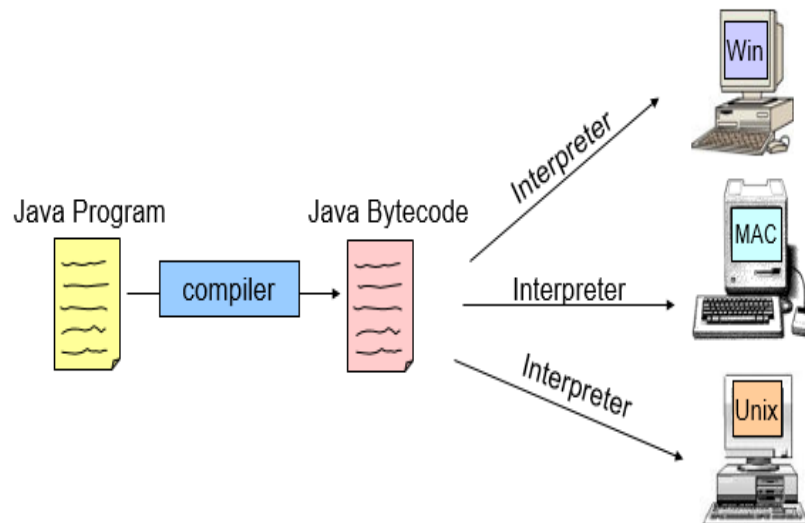
Multiple Compilers

Because different operating systems (Windows, Macs, Unix) require different machine code, you must compile most programming languages separately for each platform



Java interpreter

- Java is a little different.
- Java compiler produces bytecode not machine code.
- Bytecode can be run on any computer with the Java interpreter installed.



- A Java programming environment typically consists of several programs that perform different tasks required to edit, compile, and run a Java program.
- The following description will be based on the software development environment provided by Oracle, the company that owns and maintains Java.
- It is currently known as the Java Platform, Standard Edition 8.0 (Java SE 8).
- Versions of Java SE are available for various platforms, including Linux, Windows, and macOS computers. Free downloads are available at Sun's Web site at <http://www.oracle.com/technetwork/java/>.
- In some cases, the individual programs that make up the Java SE are available in a single program development environment, known as an integrated development environment (IDE).
- Some examples include Eclipse, jGrasp, and Oracle's own NetBeans IDE.

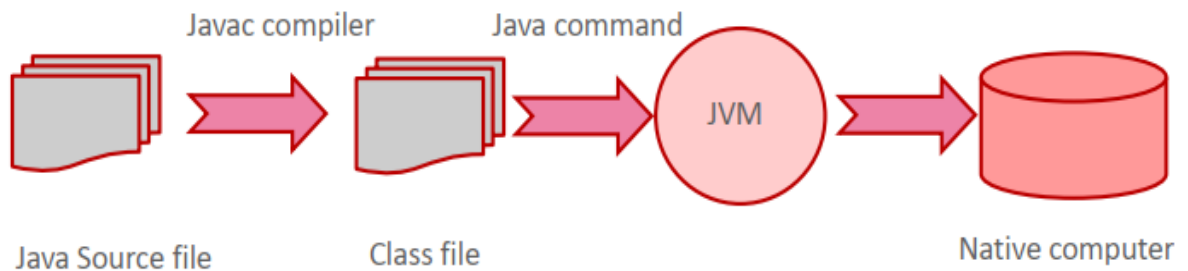
- Each of these provides a complete development package for editing, compiling, and running Java applications on a variety of platforms, including Linux, macOS, and Windows.

1.4 Overview of Java Programming:

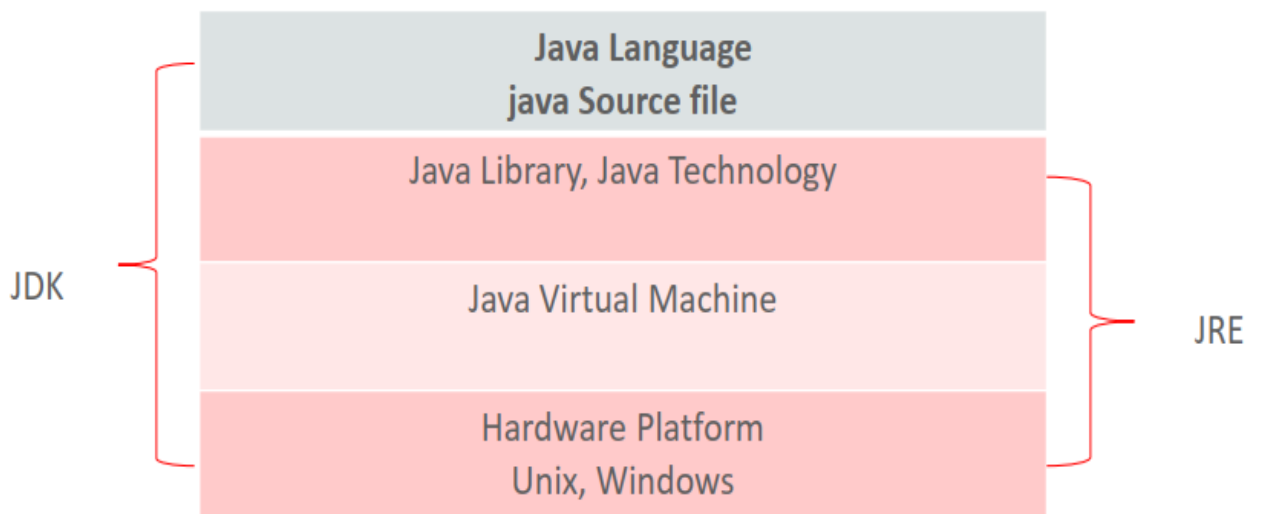
- Java is a general-purpose, class-based, object-oriented programming language designed for having lesser implementation dependencies.
- It is a computing platform for application development.
- Java is fast, secure, and reliable, therefore, It is widely used for developing Java applications in laptops, data centers, game consoles, scientific supercomputers, cell phones, etc.
- Here are important landmarks from the history of the Java language:
- The Java language was initially called OAK. Originally, it was developed for handling portable devices and set-top boxes. Oak was a massive failure.
- In 1995, Sun changed the name to “Java” and modified the language to take advantage of the burgeoning www (World Wide Web) development business.
- Later, in 2009, Oracle Corporation acquired Sun Microsystems and took ownership of three key Sun software assets: Java, MySQL, and Solaris.
- James Gosling developed the Java platform at Sun Microsystems, and the Oracle Corporation later acquired it.
- The Java Programming Language is a high-level language. Its Syntax is similar to C and C++, but it removes many of the complex , confusing features of C and C++.
- The Java Programming Language includes the feature of automatic storage management by using a garbage collector.
- The Java programming language source code is compiled into the bytecode instruction set which can be run inside the Java Virtual Machine (JVM) process.

Java Application

- In the Java Language, all of the source code is written in plain text files with the .java extension
- The java source code files are then compiled into .class extension files by the command `javac`
- A .class file contains bytecode which is a platform independent instruction set
- The java command then runs the application
- Java application translation from source code to byte code procedure.



- Oracle has two products that implement Java Platform Standard Edition, Java SE development Kit and Java SE Runtime Environment



The simplest way to learn a new language is to write a few simple example programs and execute them.

```

public class Sample
{
    public static void main (String args [])
    {
        System.out.print("Hello world of Java!");
    }
}
  
```

Step 1: create a sample source code file Sample.java

Step 2. Compile the Source code to generate the Class file

```
javac Sample.java
```

Step 3. Run the Application

```
java Sample
```

Step 4. Print out the result

Hello World of Java!

- The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.
- It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.
- The Java Virtual Machine gives runtime support to the application and can make the application independent from different hardware systems.

Chapter Two: Classes and Objects in Java

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class

2.1 Defining a class

Classes have been used since the beginning of this course. However, until now, only the most rudimentary form of a class has been used. The classes created in the preceding chapter primarily exist simply to encapsulate the **main()** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far.

Perhaps the most important thing to understand about a class is that it defines a **new data type**. Once defined, this new type can be used to create objects of that type. **Thus, a class is a template for an object, and an object is an instance of a class.** Because an object is an instance of a class, you will often see the **two words *object* and *instance* used interchangeably.**

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

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

➤ **The general form of a class definition is shown here:**

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

```
}
```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

Variables defined within a class are called **instance variables** because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as **main()**, which we have been using thus far. However, most methods will not be specified as **static** or **public**. Notice that the general form of a class does not specify a **main()** method. Java classes do not need to have a **main()** method. You only specify one if that class is the starting point for your program. Further, applets don't require a **main()** method at all.

- Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three **instance variables**: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

2.2 Creating an Object

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a **class** declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence. To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.

Again, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.

Here is a complete program that uses the Box class:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
    double width;
    double height;
    double depth;
}

// This class declares an object of type Box.
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main()** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

Volume is 3000.0

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. **For example, the following program declares two Box objects:**

```
// This program declares two Box objects.
```

```
class Box {
    double width;
    double height;
    double depth;
}
class BoxDemo2 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        // assign different values to mybox2's instance variables
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

The output produced by this program is shown here:

Volume is 3000.0

Volume is 162.0

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

2.3 Instantiating and using objects

In Java, instantiation mean to call the constructor of a class that creates an instance or object of the type of that class. In other words, creating an object of the class is called instantiation. It occupies the initial memory for the object and returns a reference. An object instantiation in Java provides the blueprint for the class. As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process.

- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.

This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

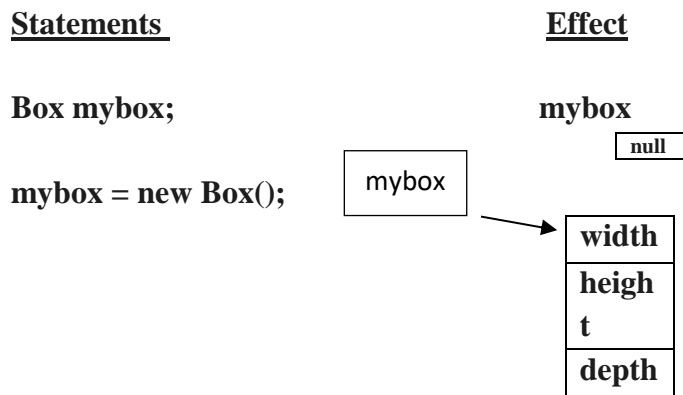
```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
```

```
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object. The effect of these two lines of code is depicted as follows:



Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers. This suspicion is, essentially, correct. An object reference is similar to a memory pointer. The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object.

It has this general form:

```
class-var = new classname( );
```

Here, **class-var** is a variable of the class type being created. The **classname** is the name of the class that is being instantiated. The class name followed by parentheses specifies the **constructor** for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors. At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's simple types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the simple types. By not applying the same overhead to the simple types that applies to objects, Java can implement the simple types more efficiently. Later, you will see object versions of the simple types that are available for your use in those situations in which complete objects of these types are needed.

It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. For the sample programs in this course, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write. Let's once again review the distinction between a **class** and an **object**. **A class creates** a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, **an object occupies space in memory**.) It is important to keep this distinction clearly in mind.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
```

```
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be **wrong**. Instead, after this fragment executes, **b1** and **b2** will both refer to the **same object**. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object. This situation is depicted here:

Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**.

For example:

```
Box b1 = new Box();
```

```
Box b2 = b1;
```

```
// ...
```

```
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

- *When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.*

2.4 Methods (Functions)

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and again.

It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it. The most important method in Java is the `main()` method

As mentioned at the beginning of this chapter, classes usually consist of two things: **instance variables and methods**. The topic of methods is a large one because Java gives them so much power and flexibility. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

There are two types of methods in Java: **predefined** and **user defined**.

In Java, predefined methods are the method that is already defined in the Java class libraries.

It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **`length()`**, **`equals()`**, **`compareTo()`**, **`sqrt()`**, **`max()`** etc.

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

Example

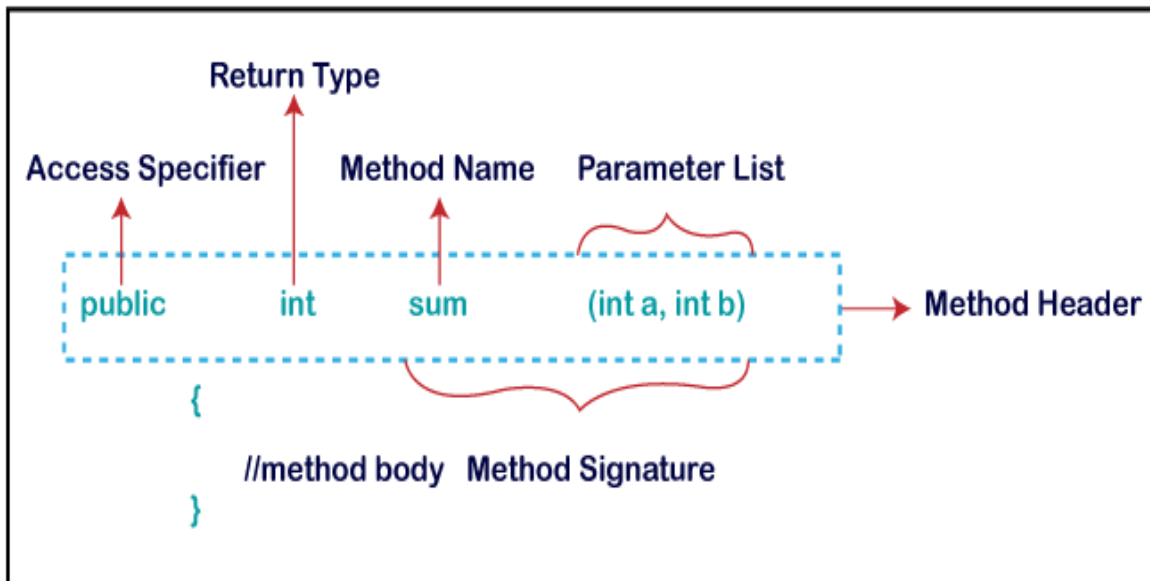
```
public class Demo {  
public static void main(String[] args) {  
// using the max() method of Math class  
System.out.print("The maximum number is: " + Math.max(20,10));  
}  
}
```

output The maximum number is: 20

This is the general form of a method:

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments.

Method Declaration



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

Here, **type** specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The **name** of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.

The **parameter-list** is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

- **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction()**. A method is invoked by its name.

- **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

return *value*;

Here, *value* is the value returned.

Adding a Method to the Box Class

Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside the box class.
class Box {
    double width;
    double height;
    double depth;

    // display volume of a box
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's instance variables
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // display volume of first box
        mybox1.volume();
```

```
        // display volume of second box
    mybox2.volume();
    }
}
```

This program generates the following output, which is the same as the previous version.

Volume is 3000.0

Volume is 162.0

Look closely at the following two lines of code:

```
    mybox1.volume();
    mybox2.volume();
```

The first line here invokes the **volume()** method on **mybox1**. That is, it calls **volume()** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume()** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume()** displays the volume of the box defined by **mybox2**. Each time **volume()** is invoked, it displays the volume for the specified box.

If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up. When **mybox1.volume()** is executed, the Java run-time system transfers control to the code defined inside **volume()**. After the statements inside **volume()** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines.

There is something very important to notice inside the **volume()** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator. When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume()** implicitly refer to the copies of those variables found in the object that invokes **volume()**.

Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

Returning a Value

While the implementation of **volume()** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement **volume()** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.
class Box {
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        // assign different values to mybox2's instance variables
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

As you can see, when **volume()** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume()**. Thus, after

vol = mybox1.volume();

executes, the value of **mybox1.volume()** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.

- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume()** could have been used in the **println()** statement directly, as shown here:

```
System.out.println("Volume is " + mybox1.volume());
```

In this case, when **println()** is executed, **mybox1.volume()** will be called automatically and its value will be passed to **println()**.

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()  
{  
    return 10 * 10;  
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square()** much more useful.

```
int square(int i)  
{  
    return i * i;  
}
```

Now, **square()** will return the square of whatever value it is called with. That is, **square()** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;  
x = square(5); // x equals 25  
x = square(9); // x equals 81  
y = 2;  
x = square(y); // x equals 4
```

In the first call to **square()**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square()** is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight.

- A *parameter* is a variable defined by a method that receives a value when the method is called.

For example, in **square()**, **i** is a parameter.

- An *argument* is a value that is passed to a method when it is invoked.

For example, **square(100)** passes 100 as an argument. Inside **square()**, the parameter **i** receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

```
mybox1.width = 10;  
mybox1.height = 20;  
mybox1.depth = 15;
```

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimension of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized method.  
class Box {  
    double width;  
    double height;  
    double depth;  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
    // sets dimensions of box  
    void setDim(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}  
class BoxDemo5 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
        // initialize each box  
        mybox1.setDim(10, 20, 15);  
        mybox2.setDim(3, 6, 9);  
        // get volume of first box
```

```
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

As you can see, the **setDim()** method is used to set the dimensions of each box. For example, when **mybox1.setDim(10, 20, 15);** is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**.

Inside **setDim()** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

2.5 Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim()**, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor. A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately. You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim()** with a constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
// Here, Box uses a constructor to initialize the dimensions of a box.
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
```

```
    depth = 10;
}

    // compute and return volume
double volume() {
    return width * height * depth;
}
}

class BoxDemo6 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

As you can see, both **mybox1** and **mybox2** were initialized by the **Box()** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println()** statement inside **Box()** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

Before moving on, let's re-examine the **new** operator. As you know, when you allocate an object, you use the following general form:

```
class-var = new classname( );
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

new Box() is calling the **Box()** constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. The default constructor

automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the **Box()** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of **Box** defines a parameterized constructor which sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```
        /* Here, Box uses a parameterized constructor to
           initialize the dimensions of a box.
           */
class Box {
    double width;
    double height;
    double depth;
        // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
        // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
class BoxDemo7 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

```
}  
}
```

The output from this program is shown here:

Volume is 3000.0

Volume is 162.0

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the **Box()** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

The *this* Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the *this* keyword. *this* can be used inside any method to refer to the *current object*.

That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

To better understand what **this** refers to, consider the following version of **Box()**:

```
// A redundant use of this.
```

```
Box(double w, double h, double d) {  
this.width = w;  
this.height = h;  
this.depth = d;  
}
```

This version of **Box()** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box()**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the **same name** inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box()** constructor inside the **Box** class. If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables. For example, here is another version of **Box()**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
```

```
Box(double width, double height, double depth) {
```

```
this.width = width;  
this.height = height;  
this.depth = depth;  
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt.

Although **this** is of no significant value in the examples just shown, it is very useful in certain situations.

2.6 Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called **garbage collection**.

It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The **finalize()** Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize()** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an object is freed, the Java run time calls the **finalize()** method on the object.

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

```
protected void finalize( )
```

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

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class. It is important to understand that **finalize()** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize()** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

If you are familiar with C++, then you know that C++ allows you to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The **finalize()** method only approximates the function of a destructor. As you get more experienced with Java, you will see that the need for destructor functions is minimal because of Java's garbage collection subsystem.

Introducing Access Control

As you know, **encapsulation** links data with the code that manipulates it. However, encapsulation provides another important attribute: **access control**. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal

Java's access specifiers are **public**, **private**, and **protected**. Java also defines a **default access level**.

- **protected** applies only when inheritance is involved. Let's begin by defining **public** and **private**.
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code.
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.

Now you can understand why **main()** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system. **When no access specifier is used, then by default the member of a class is public within its own package**, but cannot be accessed outside of its package.

In the classes developed so far, all members of a class have used the default access mode, which is essentially public. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods which are private to a class.

An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. **Here is an example:**

```
public int i;
private double j;
private int myMethod(int a, char b) { // ...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between
public and private.
*/
class Test {
    int a;           // default access
    public int b;     // public access
    private int c;   // private access
    // methods to access c
    void setc(int i) { // set c's value
        c = i;
    }
    int getc() {      // get c's value
        return c;
    }
}
class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();
        // These are OK, a and b may be accessed directly
        ob.a = 10;
        ob.b = 20;
        // This is not OK and will cause an error
        // ob.c = 100; // Error!
        // You must access c through its methods
        ob.setc(100); // OK
        System.out.println("a, b, and c: " + ob.a + " " +
            ob.b + " " + ob.getc());
    }
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc()** and **getc()**. If you were to remove the comment symbol from the beginning of the following line,

// ob.c = 100; // Error!

then you would not be able to compile this program because of the access violation. To see how access control can be applied to a more practical example, consider the example,

```
// This class defines an integer stack that can hold 10 values.
class Stack {
    /* Now, both stck and tos are private. This means
       that they cannot be accidentally or maliciously
```



```

        altered in a way that would be harmful to the stack.
        */
private int stck[] = new int[10];
private int tos;
    // Initialize top-of-stack
Stack() {
    tos = -1;
}
    // Push an item onto the stack
void push(int item) {
    if(tos==9)
        System.out.println("Stack is full.");
    else
        stck[++tos] = item;
}
    // Pop an item from the stack
int pop() {
    if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}
}

```

As you can see, now both **stck**, which holds the stack, and **tos**, which is the index of the top of the stack, are specified as **private**. This means that they cannot be accessed or altered except through **push()** and **pop()**. Making **tos** private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the **stck** array. The following program demonstrates the improved **Stack** class. Try removing the commented-out lines to prove to yourself that the **stck** and **tos** members are, indeed, inaccessible.

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();
        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
        // these statements are not legal
    }
}

```

```
        // mystack1.tos = -2;
        // mystack2.stck[3] = 100;
    }
}
```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called before any objects exist. Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as static have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block which gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a * 4** or **12**. Then **main()** is called, which calls **meth()**, passing **42** to **x**.

The three **println()** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

*It is illegal to refer to any instance variables inside of a **static** method.*

Here is the output of the program:

Static block initialized.

x = 42

a = 3

b = 12

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method

from outside its class, you can do so using the following general form:

classname.method()

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object- reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main()**, the **static** method **callme()** and the **static** variable **b** are accessed outside of their class.

```
class StaticDemo {
    static int a = 42;
    static int b = 99;
    static void callme() {
        System.out.println("a = " + a);
    }
}

class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Here is the output of this program:

a = 42

b = 99

Introducing final

A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared. (In this usage, **final** is similar to **const** in C/C++/C#.)

For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for **final** variables. Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of **final** is described in the next chapter, when inheritance is described.

Arrays Revisited

Arrays were introduced earlier in this course, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.
class Length {
    public static void main(String args[]) {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};
        System.out.println("length of a1 is " + a1.length);
        System.out.println("length of a2 is " + a2.length);
        System.out.println("length of a3 is " + a3.length);
    }
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the **length** member to good use in many situations. For example, here is an improved version of the **Stack** class. As you might recall, the earlier versions of this class always created a ten-element stack. The following version lets you create stacks of any size. The value of **stack.length** is used to prevent the stack from overflowing.

```
// Improved Stack class that uses the length array member.
```

```
class Stack {
    private int stck[];
    private int tos;
    // allocate and initialize stack
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }
    // Push an item onto the stack
    void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
    // Pop an item from the stack
    int pop() {
        if(tos < 0) {
            System.out.println("Stack underflow.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);
        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. There are two types of nested classes: *static* and *non-static*. A static nested class is one which has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used. The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class. The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test()**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
    int outer_x = 100;
    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Output from this application is shown here:

display: outer_x = 100

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display()** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main()** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test()** method. That method creates an instance of class **Inner** and the **display()** method is called. It is important to realize that class **Inner** is known only within the scope of class **Outer**. The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**. Generalizing, a nested class is no different than any other program element: it is known only within its enclosing scope.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```
// This program will not compile.
class Outer {
    int outer_x = 100;
    void test() {
        Inner inner = new Inner();
        inner.display();
    }
    // this is an inner class
    class Inner {
        int y = 10; // y is local to Inner
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
    void showy() {
        System.out.println(y); // error, y not known here!
    }
}
class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Here, **y** is declared as an instance variable of **Inner**. Thus it is not known outside of that class and it cannot be used by **showy()**.

Although we have been focusing on nested classes declared within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a **for** loop, as this next program shows.

```
// Define an inner class within a for loop.
class Outer {
    int outer_x = 100;
    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}
```

```
}  
}  
}  
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```

The output from this version of the program is shown here.

```
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100  
display: outer_x = 100
```

While nested classes are not used in most day-to-day programming, they are particularly helpful when handling events in an applet

Chapter Three: Inheritance and Polymorphism

3.1 Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a **superclass**. The class that does the inheriting is called a **subclass**. Therefore, a **subclass** is a specialized version of a **superclass**. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

<u>Superclass</u>	<u>Subclasses</u>
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Fig. 3.1 Inheritance examples

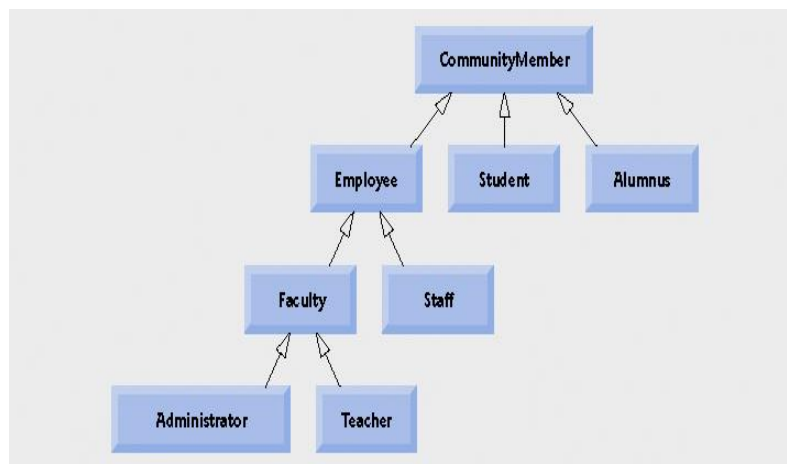


Fig. 3.2 Inheritance hierarchy for university CommunityMembers

Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a **superclass** called **A** and a **subclass** called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```

// A simple example of inheritance.
// Create a superclass.
class A {

```

```
int i, j;
void showij() {
    System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
    int k;
    void showk() {
        System.out.println("k: " + k);
    }
    void sum() {
        System.out.println("i+j+k: " + (i+j+k));
    }
}
class SimpleInheritance {
    public static void main(String args[]) {
        A superOb = new A();
        B subOb = new B();
        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();
        // The subclass has access to all public members of its superclass.
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
```

The output from this program is shown here:

Contents of superOb:

i and j: 10 20

```
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

As you can see, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij()**. Also, inside **sum()**, **i** and **j** can be referred to directly, as if they were part of **B**.

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a class declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
// body of class
}
```

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple **superclasses** into a single subclass. (This differs from C++, in which you can inherit multiple base classes.) You can, as stated, create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. **However, no class can be a superclass of itself.**

3.2 Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain private to their class. This program
contains an error and will not compile.*/
// Create a superclass.
class A {
    int i;           // public by default
    private int j;   // private to A
    void setij(int x, int y) {
        i = x;
        j = y;
    }
}
// A's j is not accessible here.
class B extends A {
    int total;
    void sum() {
        total = i + j; // ERROR, j is not accessible here
    }
}
class Access {
```

```
public static void main(String args[]) {  
    B subOb = new B();  
    subOb.setij(10, 12);  
    subOb.sum();  
    System.out.println("Total is " + subOb.total);  
}  
}
```

This program will not compile because the reference to **j** inside the **sum()** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

- *A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.*
- *A subclass can change the state of private superclass instance variables only through non-private methods provided in the superclass and inherited by the subclass.*

A More Practical Example

Let's look at a more practical example that will help illustrate the power of inheritance. Here, the final version of the **Box** class developed in the preceding chapter will be extended to include a fourth component called **weight**. Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend Box.  
class Box {  
    double width;  
    double height;  
    double depth;  
    // construct clone of an object  
    Box(Box ob) {           // pass object to constructor  
        width = ob.width;  
        height = ob.height;  
  
        depth = ob.depth;  
    }  
    // constructor used when all dimensions specified  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    // constructor used when no dimensions specified  
    Box() {  
        width = -1; // use -1 to indicate  
        height = -1; // an uninitialized  
        depth = -1; // box  
    }  
    // constructor used when cube is created  
    Box(double len) {
```

```
width = height = depth = len;
}
    // compute and return volume
double volume() {
return width * height * depth;
}
}

    // Here, Box is extended to include weight.
class BoxWeight extends Box {
double weight;    // weight of box
    // constructor for BoxWeight
BoxWeight(double w, double h, double d, double m) {
width = w;
height = h;
depth = d;
weight = m;
}
}

class DemoBoxWeight {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
}
}
```

The output from this program is shown here:

Volume of mybox1 is 3000.0

Weight of mybox1 is 34.3

Volume of mybox2 is 24.0

Weight of mybox2 is 0.076

BoxWeight inherits all of the characteristics of **Box** and adds to them the **weight** component. It is not necessary for **BoxWeight** to re-create all of the features found in **Box**. It can simply extend **Box** to meet its own purposes. **A major advantage of inheritance is that once you have created a superclass that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses.** Each subclass can precisely tailor its own classification. For example, the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box {
    int color;           // color of box
    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}
```

Remember, once you have created a superclass that defines the general aspects of an object, that superclass can be inherited to form specialized classes. Each subclass simply adds its own, unique attributes. This is the essence of inheritance.

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations.

For example, consider the following:

```
class RefDemo {
    public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;
        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " +
            weightbox.weight);
        System.out.println();
        // assign BoxWeight reference to Box reference
        plainbox = weightbox;
        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);
        /* The following statement is invalid because plainbox
           does not define a weight member. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
    }
}
```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects. Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object. It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is why **plainbox** can't access **weight** even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the **weight** field, because it does not define one.

3.3 Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box** (). Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. **super** has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

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

Here, *parameter-list* specifies any parameters needed by the constructor in the superclass. **super**() must always be the first statement executed inside a subclass' constructor.

To see how **super**() is used, consider this improved version of the **BoxWeight**() class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
    double weight; // weight of box
    // initialize width, height, and depth using super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
}
```

Here, **BoxWeight**() calls **super**() with the parameters **w**, **h**, and **d**. This causes the **Box**() constructor to be called, which initializes **width**, **height**, and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired. In the preceding example, **super**() was called with three arguments. Since constructors can be overloaded, **super**() can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, **super**() is called using the appropriate arguments. Notice that **width**, **height**, and **depth** have been made private within **Box**.

```
// A complete implementation of BoxWeight.
class Box {
    private double width;
    private double height;
```

```
private double depth;
    // construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

    // constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}

    // constructor used when cube is created
Box(double len) {
width = height = depth = len;
}

    // compute and return volume
double volume() {
return width * height * depth;
}
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
    // construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}

    // constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}

    // default constructor
BoxWeight() {
super();
weight = -1;
}
}
```



```
// constructor used when cube is created
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}
class DemoSuper {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        System.out.println();
        vol = mybox3.volume();
        System.out.println("Volume of mybox3 is " + vol);
        System.out.println("Weight of mybox3 is " + mybox3.weight);
        System.out.println();
        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
        System.out.println("Weight of myclone is " + myclone.weight);
        System.out.println();
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
        System.out.println("Weight of mycube is " + mycube.weight);
        System.out.println();
    }
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
```

Weight of mycube is 2.0

Pay special attention to this constructor in BoxWeight():

```
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
    super(ob);
    weight = ob.weight;
}
```

Notice that **super()** is called with an object of type **BoxWeight**—not of type **Box**. This still invokes the constructor **Box(Box ob)**. As mentioned earlier, a superclass variable can be used to reference any object derived from that class. Thus, we are able to pass a **BoxWeight** object to the **Box** constructor. Of course, **Box** only has knowledge of its own members.

Let's review the key concepts behind **super()**. When a subclass calls **super()**, it is calling the constructor of its immediate superclass. Thus, **super()** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super()** must always be the first statement executed inside a subclass constructor.

- A compilation error occurs if a subclass constructor calls one of its superclass constructors with arguments that do not match exactly the number and types of parameters specified in one of the superclass constructor declarations.

A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.

This usage has the following general form:

super.member

Here, *member* can be either a method or an instance variable. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A {
    int i;
}
// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the i in A
    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }
    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}
class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);
    }
}
```

```
subOb.show();
}
}
```

This program displays the following:

i in superclass: 1

i in subclass: 2

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

protected Members

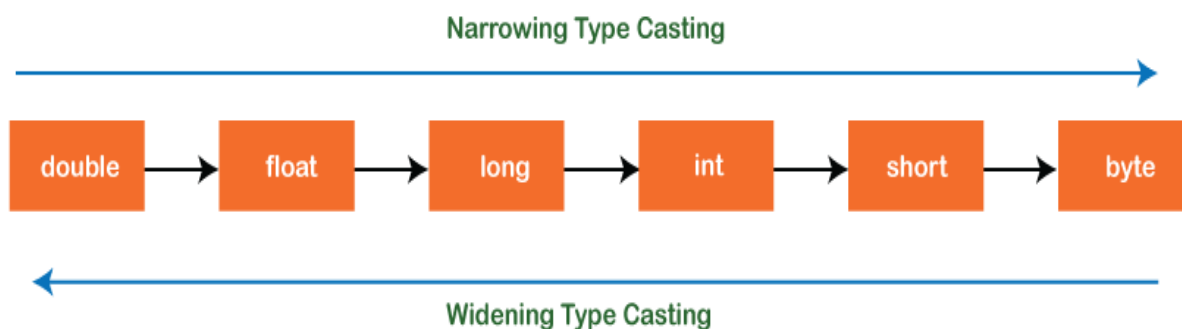
A class's **public** members are accessible wherever the program has a reference to an object of that class or one of its **subclasses**. A class's **private** members are accessible only from within the class itself. A **superclass's private** members are not inherited by its subclasses.

Using **protected** access offers an intermediate level of access between public and private. A superclass's protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package (i.e., protected members also have package access). All public and protected superclass members retain their original access modifier when they become members of the subclass (i.e., public members of the superclass become public members of the subclass, and protected members of the superclass become protected members of the subclass).

Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names. When a subclass method overrides a superclass method, the superclass method can be accessed from the subclass by preceding the

3.4 Casting

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



Type Casting in Java

- i. **Narrowing Type Casting:** converting a higher data type into a lower one. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

```
public class NarrowingTypeCasting {
    public static void main(String args[]) {
        double d = 12.76;
        //converting double data type into long data type
        long l = (long)d;
        //converting long data type into int data type
        int i = (int)l;
        System.out.println("Before conversion: "+d);
        //fractional part lost
        System.out.println("After conversion into long type: "+l);
        //fractional part lost
        System.out.println("After conversion into int type: "+i);
    }
}
```

Output

Before conversion: 12.76

After conversion into long type: 12

After conversion into int type: 12

- ii. **Widening Type Casting:** converting a lower data type into a higher one. It is also known as **implicit conversion** or **casting down**. It is done **automatically**. It is safe because there is no chance to lose data. It takes place when:
- Both data types must be compatible with each other.
 - The target type must be larger than the source type.

byte -> short -> char -> int -> long -> float -> double

For example, the conversion between **numeric** data type to **char** or **Boolean** is not done automatically.

Also, the char and Boolean data types are not compatible with each other.

```
public class WideningTypeCasting {
    public static void main(String[] args) {
        int x = 8;
        //automatically converts the integer type into long type
        long y = x;
        //automatically converts the long type into float type
        float z = y;
        System.out.println("Before conversion, int value "+x);
        System.out.println("After conversion, long value "+y);
        System.out.println("After conversion, float value "+z);
    }
}
```

output

Before conversion, the value is: 8
After conversion, the long value is: 8
After conversion, the float value is: 8.0

3.5 Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.
- Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

3.6 Method Overriding and Overloading

3.6.1 Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as **method overriding**.

Usage of Java Method Overriding

- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

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

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Consider the following:

```
// Method overriding.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
```

```
class B extends A {  
  
    int k;  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
    // display k – this overrides show() in A  
    void show() {  
        System.out.println("k: " + k);  
    }  
}  
class Override {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2, 3);  
        subOb.show(); // this calls show() in B  
    }  
}
```

The output produced by this program is shown here:

k: 3

When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**.

If you wish to access the superclass version of an overridden function, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show()** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {  
    int k;  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
    void show() {  
        super.show(); // this calls A's show()  
        System.out.println("k: " + k);  
    }  
}
```

If you substitute this version of A into the previous program, you will see the following output:

i and j: 1 2

k: 3

Here, **super.show()** calls the superclass version of **show()**.

- Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.
- The overridden method can widen the accessibility but not narrow it, i.e if it is private in the base class, the child class can make it public but not vice verse(i.e it is error if we have public in base class and private in child class)

For example, consider this modified version of the preceding example:

// Methods with differing type signatures are overloaded – not overridden.

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }

    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // overload show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show("This is k: "); // this calls show() in B
        subOb.show(); // this calls show() in A
    }
}
```

The output produced by this program is shown here:

This is k: 3

i and j: 1 2

The version of **show()** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place.

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*.

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- Dynamic method dispatch is important because this is how Java implements run-time **polymorphism**.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different version of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
    void callme() {
        System.out.println("Inside A's callme method");
    }
}
class B extends A {
    // override callme()
    void callme() {
        System.out.println("Inside B's callme method");
    }
}
class C extends A {
    // override callme()
    void callme() {
        System.out.println("Inside C's callme method");
    }
}
class Dispatch {
    public static void main(String args[]) {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C
        A r;           // obtain a reference of type A
        r = a;          // r refers to an A object
        r.callme();     // calls A's version of callme
        r = b;          // r refers to a B object
        r.callme();     // calls B's version of callme

        r = c;          // r refers to a C object
        r.callme();     // calls C's version of callme
    }
}
```


The output from the program is shown here:

Inside A's callme method
Inside B's callme method
Inside C's callme method

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme()** declared in **A**. Inside the **main()** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared.

The program then assigns a reference to each type of object to **r** and uses that reference to invoke **callme()**. As the output shows, the version of **callme()** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme()** method.

- *Readers familiar with C++ or C# will recognize that overridden methods in Java are similar to virtual functions in those languages.*

Why Overridden Methods?

- As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism. Part of the key to successfully applying polymorphism is understanding that the super classes and subclasses form a hierarchy which moves from lesser to greater specialization.

Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Applying Method Overriding

Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of various two-dimensional objects. It also defines a method called **area()** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area()** so that it returns the area of a rectangle and a triangle, respectively.

// Using run-time polymorphism.

```
class Figure {  
    double dim1;
```

```
double dim2;
Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
}
double area() {
    System.out.println("Area for Figure is undefined.");
    return 0;
}
}
class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }
    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
class Triangle extends Figure {

    Triangle(double a, double b) {
        super(a, b);
    }
    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
```

```
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area()**. The interface to this operation is the same no matter what type of figure is being used.

3.6.2 Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be **overloaded**, and the process is referred to as **method overloading**. Method overloading is one of the ways that Java implements **polymorphism**. When an overloaded method is invoked, Java uses **the type and/or number of arguments** as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}

// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}

// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
```

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

This program generates the following output:

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

As you can see, **test()** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test()** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution.

For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Overload test for two integer parameters.
    void test(int a, int b) {
```

```
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i);           // this will invoke test(double)
ob.test(123.2);       // this will invoke test(double)
}
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

As you can see, this version of **OverloadDemo** does not define **test(int)**. Therefore, when **test()** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**. Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.

Method overloading supports **polymorphism** because it is one way that Java implements the “one interface, multiple methods” paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs()** returns the absolute value of an integer, **labs()** returns the absolute value of a long integer, and **fabs()** returns the absolute value of a floating-point value. Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute

value method can use the same name. Indeed, Java's standard class library includes an absolute value method, called **abs()**. This method is overloaded by Java's **Math** class to handle all numeric types. Java determines which version of **abs()** to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **abs** represents the *general action* which is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity. When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two

operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's consider the **Box** class. Following is the latest version of **Box**:

```
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

As you can see, the **Box()** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box()** constructor. For example, the following statement is currently invalid:

Box ob = new Box();

Since **Box()** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you. Fortunately, **the solution to these problems is quite easy: simply overload the Box constructor so that it handles the situations just described.** Here is a program that contains an improved version of **Box** that does just that:

```
/* Here, Box defines three constructors to initialize the dimensions of a box various
ways.
*/
class Box {
    double width;
    double height;
    double depth;

    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }

    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }

    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons {
```

```
public static void main(String args[]) {  
    // create boxes using the various constructors  
    Box mybox1 = new Box(10, 20, 15);  
    Box mybox2 = new Box();  
    Box mycube = new Box(7);  
    double vol;  
    // get volume of first box  
    vol = mybox1.volume();  
    System.out.println("Volume of mybox1 is " + vol);  
    // get volume of second box  
    vol = mybox2.volume();  
    System.out.println("Volume of mybox2 is " + vol);  
  
    // get volume of cube  
    vol = mycube.volume();  
    System.out.println("Volume of mycube is " + vol);  
}  
}
```

The output produced by this program is shown here:

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of mycube is 343.0

As you can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

Using Objects as Parameters

So far we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.  
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
    // return true if o is equal to the invoking object  
    boolean equals(Test o) {  
        if(o.a == a && o.b == b) return true;  
        else return false;  
    }  
}
```



```
class PassOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(100, 22);  
        Test ob2 = new Test(100, 22);  
        Test ob3 = new Test(-1, -1);  
        System.out.println("ob1 == ob2: " + ob1.equals(ob2));  
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));  
    }  
}
```

This program generates the following output:

ob1 == ob2: true

ob1 == ob3: false

As you can see, the **equals()** method inside **Test** compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equals()** specifies **Test** as its type. Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types. One of the most common uses of object parameters involves constructors. Frequently you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of **Box** allows one object to initialize another:

// Here, Box allows one object to initialize another.

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // construct clone of an object  
    Box(Box ob) { // pass object to constructor  
        width = ob.width;  
        height = ob.height;  
        depth = ob.depth;  
    }  
    // constructor used when all dimensions specified  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
    // constructor used when no dimensions specified  
    Box() {
```

```
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}

    // constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons2 {
public static void main(String args[]) {
    // create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1);
double vol;
    // get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
    // get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
    // get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

As you will see when you begin to create your own classes, providing many forms of constructor methods is usually required to allow objects to be constructed in a convenient and efficient manner.

A Closer Look at Argument Passing

In general, there are **two ways** that a computer language can pass an argument to a subroutine.

1. **Call-by-value:-** This method copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
2. **Call-by-reference:-** In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed. In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example, consider the following program:

```
// Simple types are passed by value.
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();
        int a = 15, b = 20;
        System.out.println("a and b before call: " + a + " " + b);
        ob.meth(a, b);
        System.out.println("a and b after call: " + a + " " + b);
    }
}
```

The output from this program is shown here:

a and b before call: 15 20

a and b after call: 15 20

As you can see, the operations that occur inside **meth()** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of **call-by-reference**. Changes to the object inside the method *do* affect the object used as an argument.

For example, consider the following program:

```
// Objects are passed by reference.
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    // pass an object
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}
class CallByRef {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
        ob.meth(ob);
        System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
    }
}
```

This program generates the following output:

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

As you can see, in this case, the actions inside **meth()** have affected the object used as an argument. As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

- *When a simple type is passed to a method, it is done by use of call-by-value. Objects are passed by use of call-by-reference.*

Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen()** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
    int a;
    Test(int i) {
        a = i;
    }
}
```

```
Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
}

class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
            + ob2.a);
    }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

As you can see, each time **incrByTen()** is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

3.7 Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```
/* Extend BoxWeight to include shipping costs. Start with Box.*/
```

```
class Box {
private double width;
private double height;
private double depth;
    // construct clone of an object
Box(Box ob) {          // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

    // constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}

    // constructor used when no dimensions specified
Box() {
width = -1;    // use -1 to indicate
height = -1;   // an uninitialized
depth = -1;    // box
}

    // constructor used when cube is created
Box(double len) {
width = height = depth = len;
}

    // compute and return volume
double volume() {
return width * height * depth;
}
}

    // Add weight.
class BoxWeight extends Box {
double weight;        // weight of box
    // construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}

    // constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d);    // call superclass constructor
weight = m;
}

    // default constructor
BoxWeight() {
```

```
super();
weight = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}

// Add shipping costs
class Shipment extends BoxWeight {
double cost;

// construct clone of an object
Shipment(Shipment ob) { // pass object to constructor
super(ob);
cost = ob.cost;
}

// constructor when all parameters are specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); // call superclass constructor
cost = c;
}

// default constructor
Shipment() {
super();
cost = -1;
}

// constructor used when cube is created
Shipment(double len, double m, double c) {
super(len, m);
cost = c;
}
}

class DemoShipment {
public static void main(String args[]) {
Shipment shipment1 =
new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =
new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "
+ shipment1.weight);
}
```

```
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println();
vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is "
+ shipment2.weight);
System.out.println("Shipping cost: $" + shipment2.cost);
}
}
```

The output of this program is shown here:

Volume of shipment1 is 3000.0

Weight of shipment1 is 10.0

Shipping cost: \$3.41

Volume of shipment2 is 24.0

Weight of shipment2 is 0.76

Shipping cost: \$1.28

Because of inheritance, **Shipment** can make use of the previously defined classes of **Box** and **BoxWeight**, adding only the extra information it needs for its own, specific application. This is part of the value of inheritance; it allows the reuse of code.

This example illustrates one other important point: **super()** always refers to the constructor in the closest superclass. The **super()** in **Shipment** calls the constructor in **BoxWeight**. The **super()** in **BoxWeight** calls the constructor in **Box**. In a class hierarchy, if a superclass constructor requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a subclass needs parameters of its own.

*In the preceding program, the entire class hierarchy, including **Box**, **BoxWeight**, and **Shipment**, is shown all in one file. This is for your convenience only. In Java, all three classes could have been placed into their own files and compiled separately. In fact, using separate files is the norm, not the exception, in creating class hierarchies.*

When Constructors Are Called

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called **B** and a superclass called **A**, is **A**’s constructor called before **B**’s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since **super()** must be the first statement executed in a subclass’ constructor, this order is the same whether or not **super()** is used. If **super()** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// Create a super class.

class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}

// Create a subclass by extending class A.
class B extends A {
```



```
B() {  
System.out.println("Inside B's constructor.");  
}  
}  
  
// Create another subclass by extending B.  
class C extends B {  
C() {  
System.out.println("Inside C's constructor.");  
}  
}  
class CallingCons {  
public static void main(String args[]) {  
  
C c = new C();  
}  
}
```

The output from this program is shown here:

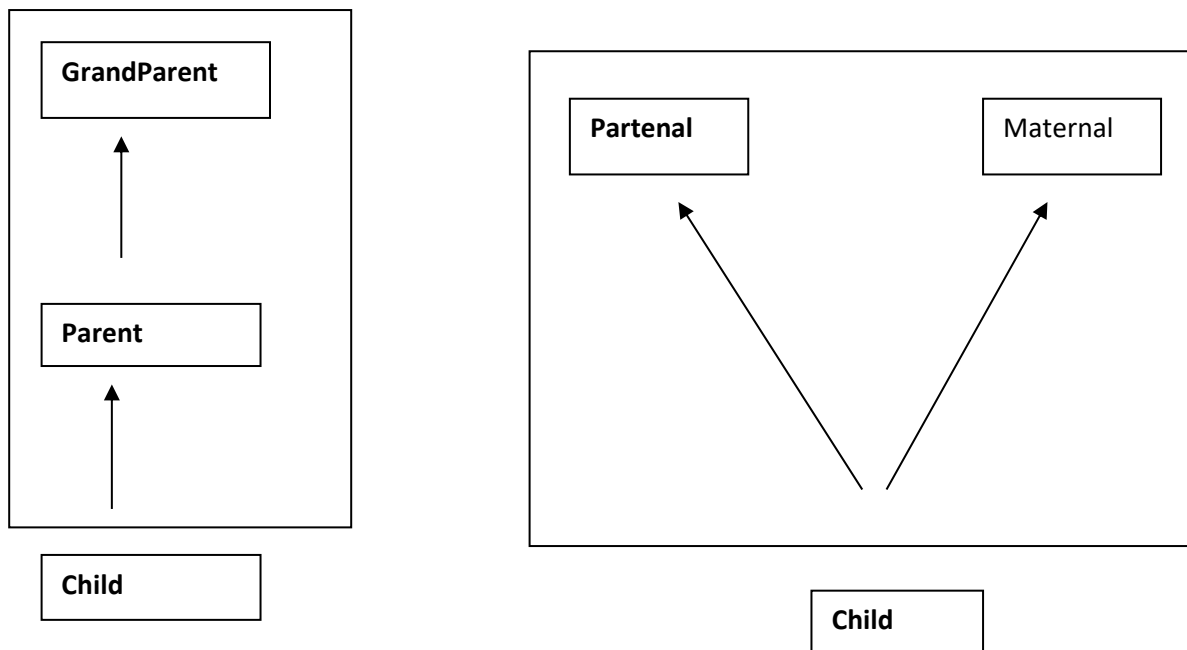
Inside A's constructor

Inside B's constructor

Inside C's constructor

As you can see, the constructors are called in order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

➤ **Multi-level** inheritance is allowed in Java but not **multiple** inheritance



Multi-level Inheritance Allowed in Java

Multiple Inheritance Not Allowed in Java

3.8 Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area()** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods which must be overridden by the subclass in order for the subclass to have any meaning.

Consider the class **Triangle**. It has no meaning if **area()** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods.

Java's solution to this problem is the *abstract method*. You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

To declare an abstract method, use this general form:

`abstract type name(parameter-list);`

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

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();
    // concrete methods are still allowed in abstract classes
    void callmetoo() {
```

```

System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}

```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo()**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit. Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

You will see this feature put to use in the next example. Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area()** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area()**.

```

// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
}
}

```

```

    return dim1 * dim2;
}
}
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }
    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Figure figref;        // this is OK, no object is created
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
    }
}

```

As the comment inside **main()** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area()**. To prove this to yourself, try creating a subclass that does not override **area()**.

You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Using final with Inheritance

The keyword **final** has three uses. **First**, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The **other two uses of final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.

- Methods declared as **final** cannot be overridden.

The following fragment illustrates final:

```

class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}
class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}

```

Because **meth()** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*. However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```

final class A {
    // ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    // ...
}

```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

3.9 The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array. **Object** defines the following methods, which means that they are available in every object.

Method

Purpose

Object clone() Creates a new object that is the same as the object being cloned.

boolean equals(Object object) Determines whether one object is equal to another.

<code>void finalize()</code>	Called before an unused object is recycled.
<code>Class getClass()</code>	Obtains the class of an object at run time.
<code>int hashCode()</code>	Returns the hash code associated with the invoking object.
<code>void notify()</code>	Resumes execution of a thread waiting on the invoking object.
<code>void notifyAll()</code>	Resumes execution of all threads waiting on the invoking object.
<code>String toString()</code>	Returns a string that describes the object.
<code>void wait()</code>	Waits on another thread of execution.
<code>void wait(long <i>milliseconds</i>)</code>	
<code>void wait(long <i>milliseconds</i>, int <i>nanoseconds</i>)</code>	

The methods **getClass()**, **notify()**, **notifyAll()**, and **wait()** are declared as **final**, and are related to multithreading.

Two methods: **equals()** and **toString()**.

- The **equals()** method compares the contents of two objects. It returns **true** if the objects are equivalent, and **false** otherwise.

The **toString()** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println()**. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

3.10 Interfaces

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

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body.

It is used to achieve abstraction and multiple inheritance in Java.

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

Java Interface also represents the IS-A relationship



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

Summary

- We use the abstract keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.

- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class

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

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

Output:

```
I am a
I am b
I am c
I am d
```

Chapter 4: Exception Handling

4.1 Exceptions Overview

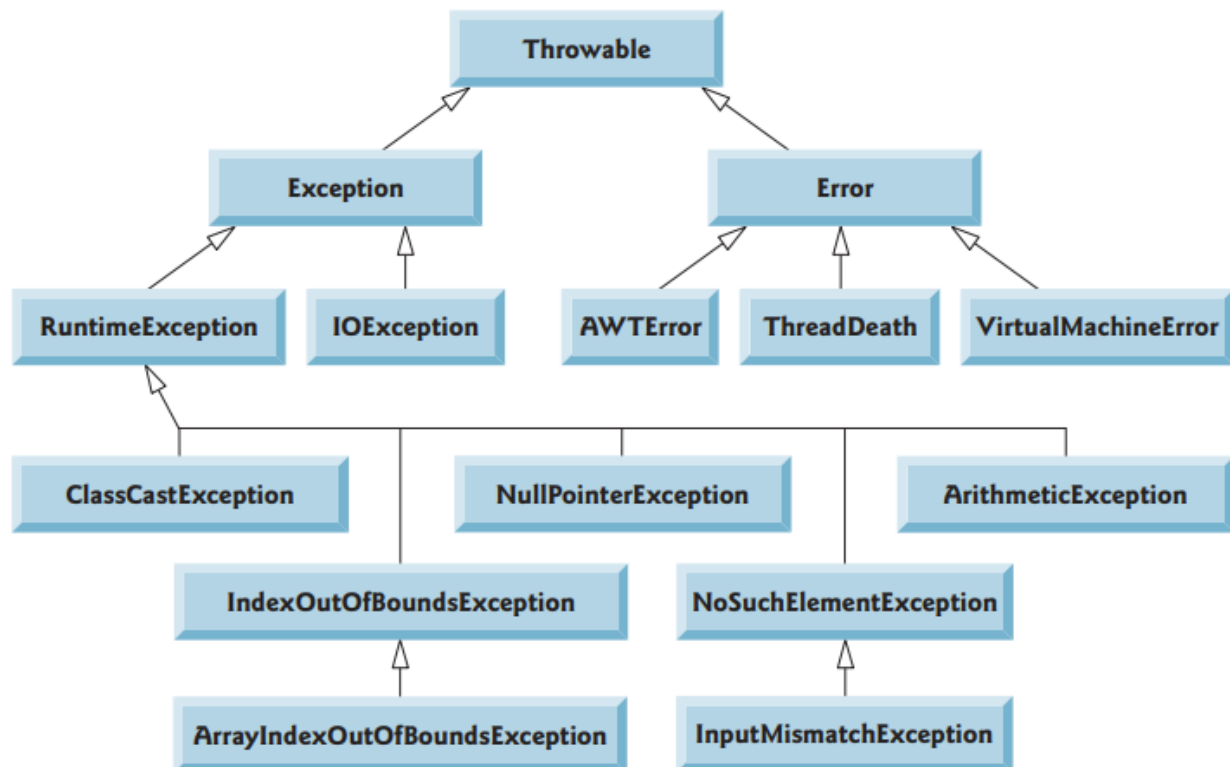
What is a problem that arises during the execution of a program that disrupts the program flow and may cause a program to fail or terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

Some examples are:

- Accessing an out-of-bounds array element
- Performing illegal arithmetic
- Illegal arguments to methods
- Hardware failures
- Writing to a read-only file

4.2 Hierarchy of Java Exception classes

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



4.2.1 Types of Java Exceptions

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

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

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

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

3) Error

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

4.3 Catching Exceptions

Catching or handling Exception: is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained. It handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. **Exception handling** enables you to create applications that can resolve (or handle) exceptions. In many cases, handling an exception allows a program to continue executing as if no problem had been encountered. The features presented here help you write robust and fault-tolerant programs that can deal with problems and continue executing or terminate gracefully. First, we demonstrate basic exception-handling techniques by handling an exception that occurs when a method attempts to divide an integer by zero. Next, we introduce several classes at the top of Java's exception-handling class hierarchy. As you'll see, only classes that extend Throwable (package java.lang) directly or indirectly can be used with exception handling.

Java Exception Keywords

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

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

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

JavaExceptionExample.java

```
public class JavaExceptionExample{
    public static void main(String args[]){
        try{
            //code that may raise exception
            int data=100/0;
        }catch(ArithmeticException e){System.out.println(e);}
        //rest code of the program
        System.out.println("rest of the code...");
    }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

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

4.3.1 Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method. If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

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

Syntax of Java try-catch

1. **try**{
2. //code that may throw an exception
3. **}catch**(Exception_class_Name ref){ }

Syntax of try-finally block

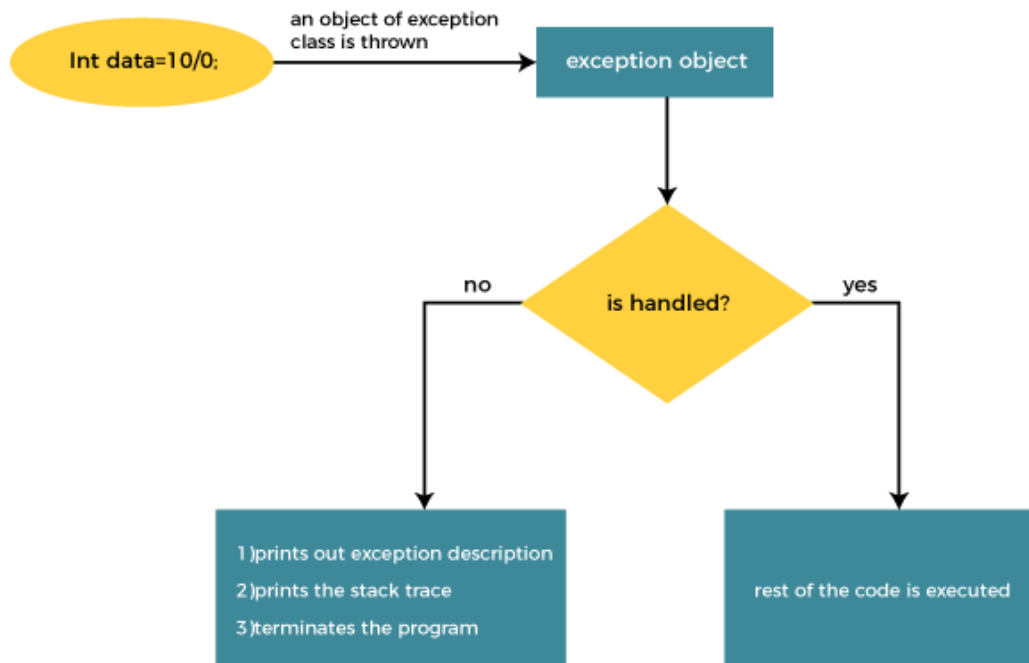
1. **try**{
2. //code that may throw an exception
3. **}finally**{ }

Java catch block

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

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

Internal Working of Java try-catch block



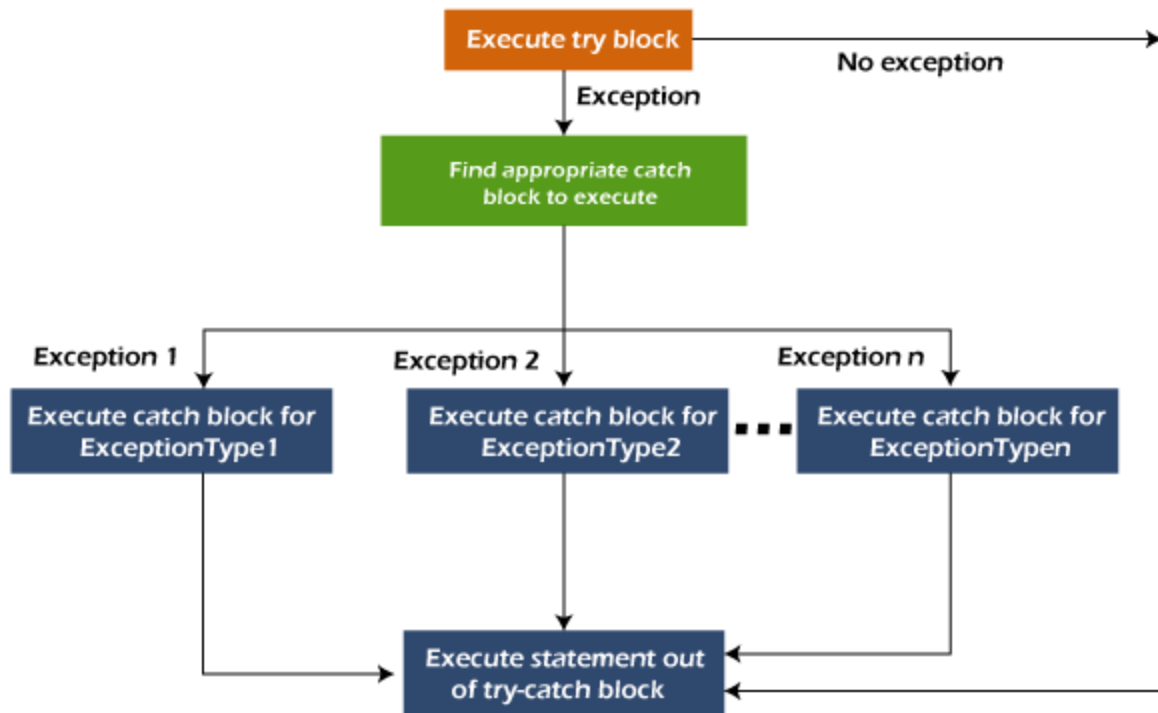
The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

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

4.3.2 Java Multi-catch block

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

Flowchart of Multi-catch Block



Example 1

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

MultipleCatchBlock1.java

```

public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}
  
```

}

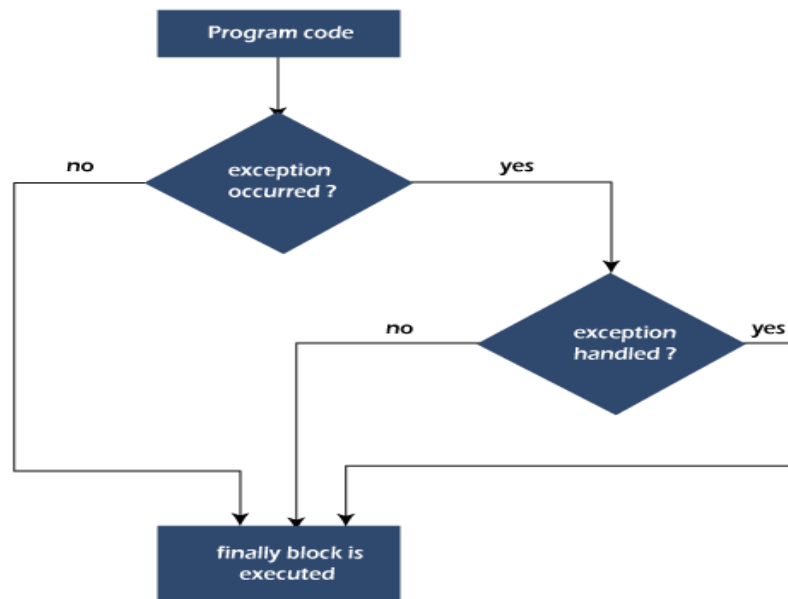
Output:

Arithmetic Exception occurs

rest of the code

4.3.3 Finally Block

Java finally block is a block used to execute important code such as closing the connection, etc. Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not. The finally block follows the try-catch block. Flowchart of finally block



Why use Java finally block?

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

TestFinallyBlock.java

```

class TestFinallyBlock {
    public static void main(String args[]){
        try{
            //below code do not throw any exception
            int data=25/5;
            System.out.println(data);
        }
        //catch won't be executed
        catch(NullPointerException e){
            System.out.println(e);
        }
    }
}
  
```

```
//executed regardless of exception occurred or not
finally {
    System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
```

4.3.4 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.

4.3.5 Throws Keyword

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

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

Syntax of Java throws

```
return_type method_name() throws exception_class_name{  
    //method code  
}
```

Which exception should be declared?

Ans: Checked exception only, because:

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

Advantage of Java throws keyword

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

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

Java throws Example

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

Testthrows1.java

```
import java.io.IOException;  
class Testthrows1{  
    void m()throws IOException{  
        throw new IOException("device error");//checked exception  
    }  
    void n()throws IOException{  
        m();  
    }  
    void p(){  
        try{  
            n();  
        }catch(Exception e){System.out.println("exception handled");}  
    }  
    public static void main(String args[]){  
        Testthrows1 obj=new Testthrows1();  
        obj.p();  
        System.out.println("normal flow...");  
    }  
}
```

4.4 Errors and Runtime Exceptions

Exceptions and errors both are subclasses of Throwable class. The error indicates a problem that mainly occurs due to the lack of system resources and our application should not catch these types of problems. Some of the examples of errors are system crash error and out of memory error. Errors mostly occur at runtime that's they belong to an unchecked type.

Exceptions are the problems which can occur at runtime and compile time. It mainly occurs in the code written by the developers. Exceptions are divided into two categories such as checked exceptions and unchecked exceptions.

Sr. No.	Key	Error	Exception
1	Type	Classified as an unchecked type	Classified as checked and unchecked
2	Package	It belongs to java.lang.error	It belongs to java.lang.Exception
3	Recoverable/ Irrecoverable	It is irrecoverable	It is recoverable
4		It can't be occur at compile time	It can occur at run time compile time both
5	Example	OutOfMemoryError ,IOError	NullPointerException , SQLException

Example of Error

```
public class ErrorExample {
    public static void main(String[] args){
        recursiveMethod(10)
    }
    public static void recursiveMethod(int i){
        while(i!=0){
            i=i+1;
            recursiveMethod(i);
        }
    }
}
```

Output

Exception in thread "main" java.lang.StackOverflowError
at ErrorExample.ErrorExample(Main.java:42)

Example of Exception

```
public class ExceptionExample {
    public static void main(String[] args){
        int x = 100;
        int y = 0;
        int z = x / y;
    }
}
```

Output

```
java.lang.ArithmeticException: / by zero
at ExceptionExample.main(ExceptionExample.java:7)
```


Chapter 5: Packages

5.1 Packages

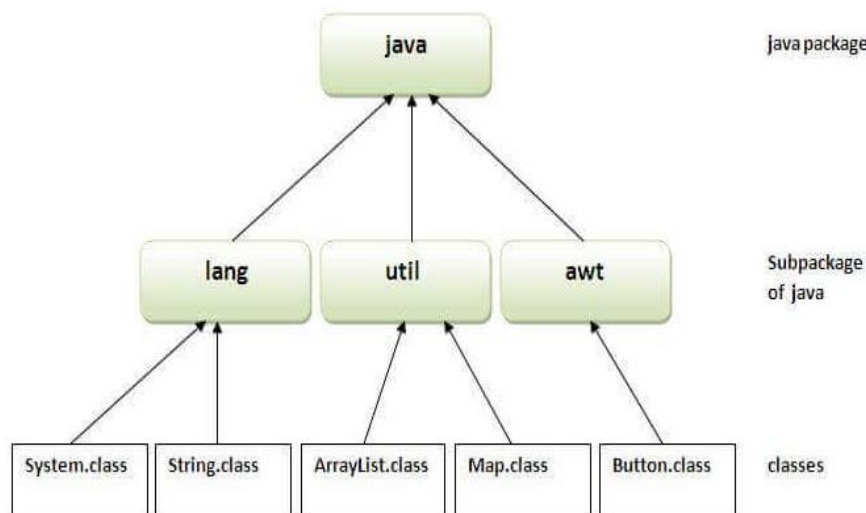
A java package is a group of similar types of classes, interfaces and sub-packages. A package is organized together under a single *namespace*

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc. Here, we will have the detailed learning of creating and using user-defined packages. The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

❖ Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

5.2 The import Statement

To use a built-in class or a package from the library, you need to use the **import** keyword:

```
import package.name.Class; // Import a single class
```

```
import package.name.*; // Import the whole package
```

If you find a class you want to use, for example, the **Scanner** class, which is used to get user input, write the following code:

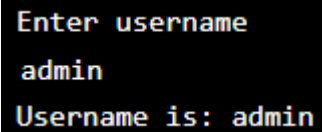
```
import java.util.Scanner;
```

In the example above, **java.util** is a package, while **Scanner** is a class of the **java.util** package. To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Example: Using the **Scanner** class to get user input:

```
import java.util.Scanner;
class MyClass {
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);
        System.out.println("Enter username");
        String userName = myObj.nextLine();
        System.out.println("Username is: " + userName);
    }
}
```

Output



```
Enter username
admin
Username is: admin
```

There are many packages to choose from. In the previous example, we used the Scanner class from the **java.util** package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the **java.util** package:

```
import java.util.*;
```

5.3 Static Imports

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Simple Example of static import

```
import static java.lang.System.*;
class StaticImportExample{
    public static void main(String args[]){
        out.println("Hello");//Now no need of System.out
        out.println("Java");
    }
}
```

Output:Hello
Java

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

5.4 Defining Packages

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

```

└─ root
    └─ mypack
        └─ MyPackageClass.java

```

To create a package, use the **package** keyword:

MyPackageClass.java

```

package mypack;
class MyPackageClass {
    public static void main(String[] args) {
        System.out.println("This is my package!");
    }
}

```

Save the file as MyPackageClass.java, and compile it

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

This forces the compiler to create the "mypack" package.

The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the MyPackageClass.java file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be: This is my package!

5.5 Package class

The package class provides methods to get information about the specification and implementation of a package. It provides methods such as `getName()`, `getImplementationTitle()`, `getImplementationVendor()`, `getImplementationVersion()` etc.

Example of Package class

In this example, we are printing the details of java.lang package by invoking the methods of package class.

```

class PackageInfo{
    public static void main(String args[]){
        Package p=Package.getPackage("java.lang");
        System.out.println("package name: "+p.getName());
        System.out.println("Specification Title: "+p.getSpecificationTitle());
        System.out.println("Specification Vendor: "+p.getSpecificationVendor());
        System.out.println("Specification Version: "+p.getSpecificationVersion());
        System.out.println("Implementaion Title: "+p.getImplementationTitle());
        System.out.println("Implementation Vendor: "+p.getImplementationVendor());
    }
}

```

```
System.out.println("Implementation Version: "+p.getImplementationVersion());
System.out.println("Is sealed: "+p.isSealed());
}
}
```

Output:package name: java.lang

Specification Title: Java Platform API Specification

Specification Vendor: Sun Microsystems, Inc.

Specification Version: 1.6

Implementation Title: Java Runtime Environment

Implementation Vendor: Sun Microsystems, Inc.

Implementation Version: 1.6.0_30

IS sealed: false

Chapter 6: Data structures

Data Structure is a branch of Computer Science which studies about a way to store and organize data so that it can be used efficiently. The data structure name indicates itself that organizing the data in memory. The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

6.1 The Set

The set is an interface available in the java.util package. The set interface extends the Collection interface. An unordered collection or list in which duplicates are not allowed is referred to as a collection interface. The set interface is used to create the mathematical set. The set interface use collection interface's methods to avoid the insertion of the same elements. **SortedSet** and **NavigableSet** are two interfaces that extend the set implementation. The **NavigableSet** extends the **SortedSet**, so it will not retain the insertion order and store the data in a sorted way.

```
import java.util.*;
public class setExample{
    public static void main(String[] args)
    {
        // creating LinkedHashSet using the Set
        Set<String> data = new LinkedHashSet<String>();
        data.add("JavaProgramming");
        data.add("Set");
        data.add("Example");
        data.add("Set");
        System.out.println(data);
    }
}
```

```
David Tech@TechTips MINGW64 /e/OneDrive - Wolaita sodo university/Class/Java programming/programs
$ java setExample
[JavaProgramming, Set, Example]
```

6.2 Operations on the Set Interface

On the Set, we can perform all the basic mathematical operations like intersection, union and difference.

Suppose, we have two sets, i.e., set1 = [22, 45, 33, 66, 55, 34, 77] and set2 = [33, 2, 83, 45, 3, 12, 55]. We can perform the following operation on the Set:

Intersection: The intersection operation returns all those elements which are present in both the set. The intersection of set1 and set2 will be [33, 45, 55].

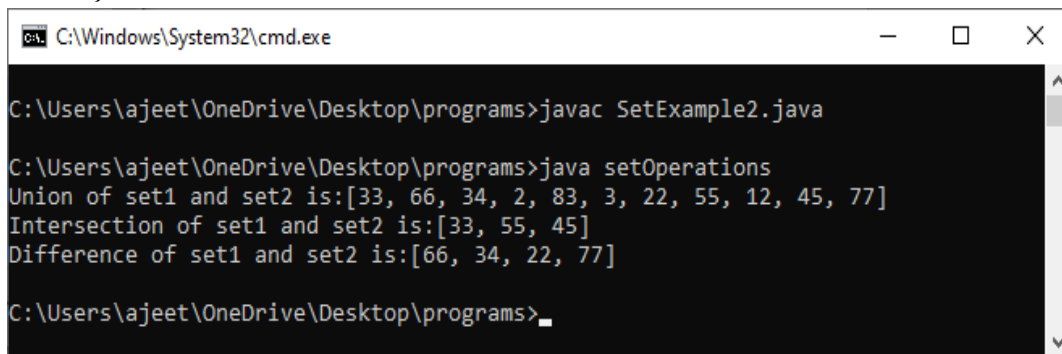
Union: The union operation returns all the elements of set1 and set2 in a single set, and that set can either be set1 or set2. The union of set1 and set2 will be [2, 3, 12, 22, 33, 34, 45, 55, 66, 77, 83].

Difference: The difference operation deletes the values from the set which are present in another set. The difference of the set1 and set2 will be [66, 34, 22, 77].

In set, **addAll()** method is used to perform the union, **retainAll()** method is used to perform the intersection and **removeAll()** method is used to perform difference. Let's take an example to understand how these methods are used to perform the intersection, union, and difference operations.

```
import java.util.*;
public class SetOperations
{
    public static void main(String args[])
    {
        Integer[] A = {22, 45, 33, 66, 55, 34, 77};
        Integer[] B = {33, 2, 83, 45, 3, 12, 55};
    }
}
```

```
Set<Integer> set1 = new HashSet<Integer>();
set1.addAll(Arrays.asList(A));
Set<Integer> set2 = new HashSet<Integer>();
set2.addAll(Arrays.asList(B));
// Finding Union of set1 and set2
Set<Integer> union_data = new HashSet<Integer>(set1);
union_data.addAll(set2);
System.out.print("Union of set1 and set2 is:");
System.out.println(union_data);
// Finding Intersection of set1 and set2
Set<Integer> intersection_data = new HashSet<Integer>(set1);
intersection_data.retainAll(set2);
System.out.print("Intersection of set1 and set2 is:");
System.out.println(intersection_data);
// Finding Difference of set1 and set2
Set<Integer> difference_data = new HashSet<Integer>(set1);
difference_data.removeAll(set2);
System.out.print("Difference of set1 and set2 is:");
System.out.println(difference_data);
}
}
```



```
C:\Windows\System32\cmd.exe
C:\Users\ajeet\OneDrive\Desktop\programs>javac SetExample2.java
C:\Users\ajeet\OneDrive\Desktop\programs>java setOperations
Union of set1 and set2 is:[33, 66, 34, 2, 83, 3, 22, 55, 12, 45, 77]
Intersection of set1 and set2 is:[33, 55, 45]
Difference of set1 and set2 is:[66, 34, 22, 77]
C:\Users\ajeet\OneDrive\Desktop\programs>
```

6.3 Set Implementation Classes

There are three general-purpose Set implementations HashSet, TreeSet, and LinkedHashSet. Which of these three to use is generally straightforward. HashSet is much faster than TreeSet (constant-time versus log-time for most operations) but offers no ordering guarantees. If you need to use the operations in the SortedSet interface, or if value-ordered iteration is required, use TreeSet; otherwise, use HashSet. It's a fair bet that you'll end up using HashSet most of the time. LinkedHashSet is in some sense intermediate between HashSet and TreeSet. Implemented as a hash table with a linked list running through it, it provides insertion-ordered iteration (least recently inserted to most recently) and runs nearly as fast as HashSet. The LinkedHashSet implementation

spares its clients from the unspecified, generally chaotic ordering provided by HashSet without incurring the increased cost associated with TreeSet.

One thing worth keeping in mind about HashSet is that iteration is linear in the sum of the number of entries and the number of buckets (the capacity). Thus, choosing an initial capacity that's too high can waste both space and time. On the other hand, choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity. If you don't specify an initial capacity, the default is 16. In the past, there was some advantage to choosing a prime number as the initial capacity. This is no longer true. Internally, the capacity is always rounded up to a power of two. The initial capacity is specified by using the int constructor. The following line of code allocates a HashSet whose initial capacity is 64.

```
Set<String> s = new HashSet<String>(64);
```

The HashSet class has one other tuning parameter called the load factor. If you care a lot about the space consumption of your HashSet, read the HashSet documentation for more information. Otherwise, just accept the default; it's almost always the right thing to do.

If you accept the default load factor but want to specify an initial capacity, pick a number that's about twice the size to which you expect the set to grow. If your guess is way off, you may waste a bit of space, time, or both, but it's unlikely to be a big problem.

Following is an example to explain Set functionality

```
import java.util.*;
public class SetDemo {
    public static void main(String args[]) {
        int count[] = {34, 22, 10, 60, 30, 22};
        Set<Integer> set = new HashSet<Integer>();
        try {
            for(int i = 0; i < 5; i++) {
                set.add(count[i]);
            }
            System.out.println(set);
            TreeSet sortedSet = new TreeSet<Integer>(set);
            System.out.println("The sorted list is:");
            System.out.println(sortedSet);
            System.out.println("The First element of the set is: " + (Integer)sortedSet.first());
            System.out.println("The last element of the set is: " + (Integer)sortedSet.last());
        }
        catch(Exception e) {}
    }
}
```

Output

```
[34, 22, 10, 60, 30]
```

```
The sorted list is:
```

```
[10, 22, 30, 34, 60]
```

```
The First element of the set is: 10
```

The last element of the set is: 60

6.4 The List

List in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list. The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming.

Syntax:

```
List<Obj> list = new ArrayList<Obj> ();
```

```
import java.util.ArrayList;
import java.util.List;
// Main class
class ListExample{
    public static void main(String args[])
    {
        // Creating an object of List interface,
        // implemented by ArrayList class
        List<String> al = new ArrayList<>();
        al.add("CS");
        al.add("IS");
        al.add(1, "Informatics");
        System.out.println(al);
    }
}
```

6.5 The Queue

The Queue interface is present in java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order.

It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.

Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.

PriorityBlockingQueue is one alternative implementation if the thread-safe implementation is needed. Since Queue is an interface, objects cannot be created of the type queue.

We always need a class which extends this list in order to create an object.

// Obj is the type of the object to be stored in Queue

```
Queue<Obj> queue = new PriorityQueue<Obj> ();
```

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample {
    public static void main(String[] args)
    {
        Queue<Integer> q= new LinkedList<>();
        for (int i = 0; i < 5; i++)
            q.add(i);
        System.out.println("Elements of queue "+ q);
        int removedele = q.remove();
        System.out.println("removed element-"+ removedele);
        System.out.println(q);
        int head = q.peek();
        System.out.println("head of queue-"+ head);
        int size = q.size();
        System.out.println("Size of queue-"+ size);
    }
}
```

- PriorityQueue class which is implemented in the collection framework provides us a way to process the objects based on the priority.
- It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play.
- Let's see how to create a queue object using this class.

```
import java.util.*;
class GfG {
    public static void main(String args[])
    {
        // Creating empty priority queue
        Queue<Integer> pQueue = new
PriorityQueue<Integer>();
        pQueue.add(10);
        pQueue.add(20);
        pQueue.add(15);
        System.out.println(pQueue.peek());
        System.out.println(pQueue.poll());
        System.out.println(pQueue.peek());
    }
}
import java.util.*;
public class PriorityExample{
    public static void main(String args[])
    {
        Queue<String> pq = new PriorityQueue<>();
        pq.add("Mikiyas");
        pq.add("Yared");
        pq.add("Yabtsega");
        System.out.println(pq);
        pq.remove(" Yabtsega");
        System.out.println(pq);
    }
}
```

Characteristics of Queue

The Queue is used to insert elements at the end of the queue and removes from the beginning of the queue. It follows FIFO concept.

LinkedList, ArrayBlockingQueue and PriorityQueue are the most frequently used implementations.

If any null operation is performed on BlockingQueues, NullPointerException is thrown.

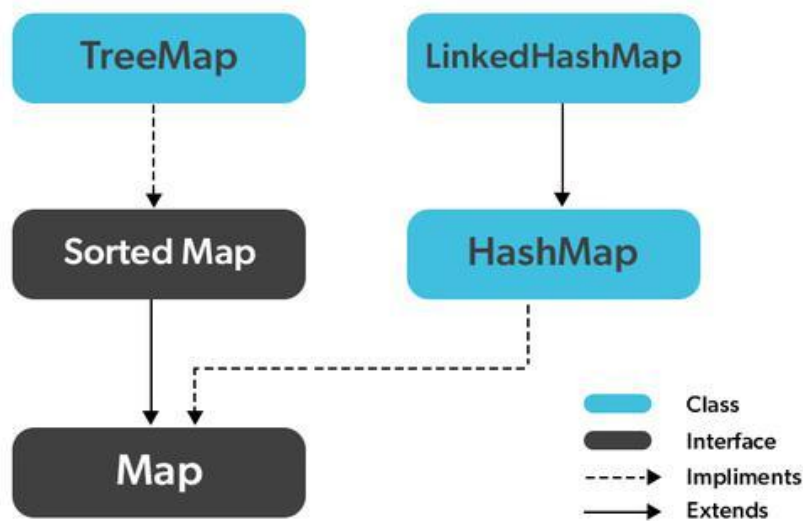
The Queues which are available in java.util package are Unbounded Queues.

The Queues which are available in java.util.concurrent package are the Bounded Queues.

All Queues except the Deques supports insertion and removal at the tail and head of the queue respectively. The Deques support element insertion and removal at both ends.

6.6 Map

- The map interface is present in `java.util` package represents a mapping between a key and a value.
- The Map interface is not a subtype of the Collection interface.
- Therefore it behaves a bit differently from the rest of the collection types. A map contains unique keys.
- Maps are perfect to use for key-value association mapping such as dictionaries.
- The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys.
- Some common scenarios are as follows:
- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).



Since Map is an interface, objects cannot be created of the type map. We always need a class that extends this map in order to create an object

```
Map hm = new HashMap();
```

```
// Obj is the type of the object to be stored in Map
```

Characteristics of a Map Interface

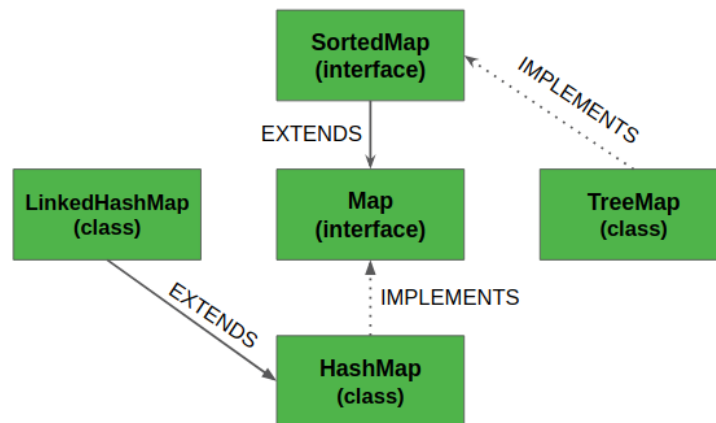
- A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.

- The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.
- There are two interfaces for implementing Map in java. They are Map and SortedMap, and three classes: HashMap, TreeMap, and LinkedHashMap.

```

class MapExample{
    // Main driver method
    public static void main(String args[])
    {
        // Creating an empty HashMap
        Map<String, Integer> hm= new HashMap<String,
Integer>();
        hm.put("a", new Integer(100));
        hm.put("b", new Integer(200));
        hm.put("c", new Integer(300));
        hm.put("d", new Integer(400));
        // Traversing through Map using for-each loop
        for (Map.Entry<String, Integer> me :hm.entrySet()) {
            System.out.print(me.getKey() + ":");
            System.out.println(me.getValue());
        }
    }
}

```



MAP Hierarchy in Java

Classes that implement the Map interface

Part Two

Java Programming Module

Chapter 1: Overview of Java programming

1.1 Introduction to Java Programming

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language. Java was developed by *Sun Microsystems* in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

Simple.java

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

1.2 Types of Java Applications:

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

1) Standalone Application

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

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, [spring](#), Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

1.3 Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

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

2) Java EE (Java Enterprise Edition)

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, [JPA](#), etc.

3) **Java ME (Java Micro Edition):** It is a micro platform that is dedicated to mobile applications.

4) **JavaFX:** It is used to develop rich internet applications. It uses a lightweight user interface API.

1.4 Variables in Java

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. Variable is a name of memory location. A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

```
int data=50;//Here data is variable
```

1) **Local Variable:** A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists. A local variable cannot be defined with "static" keyword.

2) **Instance Variable:** A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static. It is called an instance variable because its value is instance-specific and is not shared among instances.

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

1.5 Identifiers in Java

Identifiers in Java are symbolic names used for identification. They can be a class name, variable name, method name, package name, constant name, and more. There are some rules and conventions for declaring the identifiers in Java. If the identifiers are not properly declared, we may get a compile-time error. Following are some rules and conventions for declaring identifiers:

- A valid identifier must have characters [A-Z] or [a-z] or numbers [0-9], and underscore(_) or a dollar sign (\$). for example, @number is not a valid identifier because it contains a special character which is @.
- There should not be any space in an identifier. For example, **sum of number** is an invalid identifier.

- An identifier should not contain a number at the starting. For example, **123number** is an invalid identifier.
- An identifier should be of length 4-15 letters only. However, there is no limit on its length. But, it is good to follow the standard conventions.
- We can't use the Java reserved keywords as an identifier such as int, float, double, char, etc. For example, int double is an invalid identifier in Java.
- An identifier should not be any query language keywords such as SELECT, FROM, COUNT, DELETE, etc.

1.6 Data Types in Java

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

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

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Data Type	Default Value	Default size
Boolean	false	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
Double	0.0d	8 byte

Example to understand the types of variables in java

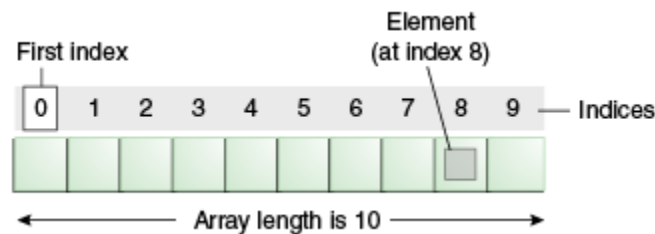
```
public class A
{
    static int m=100;//static variable
    void method()
    {
```

```
    int n=90;//local variable
}
public static void main(String args[])
{
    int data=50;//instance variable
}
} //end of class Java Variable Example: Add Two Numbers
public class Simple{
public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c); } }
```

1.7 Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location. **It** is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, and 2nd element is stored on 1st index and so on.



1.7.1 Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
 dataType []arr; (or)
2. dataType arr[];

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

int a[]={ 33,3,4,5 };//declaration, instantiation and initialization Let's see the simple example to print this array.

//Java Program to illustrate the use of declaration, instantiation

//and initialization of Java array in a single line

class Testarray1 {

public static void main(String args[]){

int a[]={ 33,3,4,5 };//declaration, instantiation and initialization

//printing array

for(**int** i=0;i<a.length;i++)//length is the property of array

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

}}

1.7.2 for-each Loop for Java Array

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

for(data_type variable:array){

//body of the loop

}

//Java Program to print the array elements using for-each loop

class Testarray1 {

public static void main(String args[]){

int arr[]={ 33,3,4,5};

//printing array using for-each loop

for(**int** i:arr)

System.out.println(i); }

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

1. **int**[][] arr=**new int**[3][3];//3 row and 3 column

Example to initialize Multidimensional Array in Java

1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;

4. arr[1][0]=4;

Example of Multidimensional Java Array

//Java Program to illustrate the use of multidimensional array

```
class Testarray3{
    public static void main(String args[]){
        //declaring and initializing 2D array
        int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
        //printing 2D array
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();
        }
    }
}
```

1.8 Java Control and Repetition Statements

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](#) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
 - if statements
 - switch statement
2. Loop statements
 - do while loop
 - while loop
 - for loop
 - for-each loop
3. Jump statements
 - break statement
 - continue statement

1.8.1 Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {  
statement 1; //executes when condition is true  
}
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

```
public class Student {  
public static void main(String[] args) {  
int x = 10;  
int y = 12;  
if(x+y > 20) {  
System.out.println("x + y is greater than 20");  
}  
}  
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The [if-else statement](#) is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {  
statement 1; //executes when condition is true  
}  
else{
```

```
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
    public static void main(String[] args) {
        int x = 10;
        int y = 12;
        if(x+y < 10) {
            System.out.println("x + y is less than 10");
        } else {
            System.out.println("x + y is greater than 20");
        } } }
```

Output:

x + y is greater than 20

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
    statement 1; //executes when condition 1 is true
}
else if(condition 2) {
    statement 2; //executes when condition 2 is true
}
else {
    statement 2; //executes when all the conditions are false
}
```

Consider the following example.

Student.java

```
public class Student {
    public static void main(String[] args) {
        double result = 79.5;
        if(result >=90&&result<=100) {
            System.out.println("A+");
        } else if (result >=80&&result<90) {
            System.out.println("A-");
        } else if(result >=75&&result<80) {
```

```
System.out.println("B");
} else {
System.out.println("C");
} } }
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
} else{
statement 2; //executes when condition 2 is false
}
}
```

Consider the following example.

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";
if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
} else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
} else {
System.out.println(address.split(",")[0]); }
} else {
System.out.println("You are not living in India");
} } }
```

Switch Statement:

In Java, [Switch statements](#) are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate

- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){  
    case value1:  
        statement1;  
        break;  
    case valueN:  
        statementN;  
        break;  
    default:  
        default statement;  
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {  
    public static void main(String[] args) {  
        int num = 2;  
        switch (num){  
            case 0:  
                System.out.println("number is 0");  
                break;  
            case 1:  
                System.out.println("number is 1");  
                break;  
            default:  
                System.out.println(num);  
        }  
    }  
}
```

Output: 2

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

1.8.2 Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

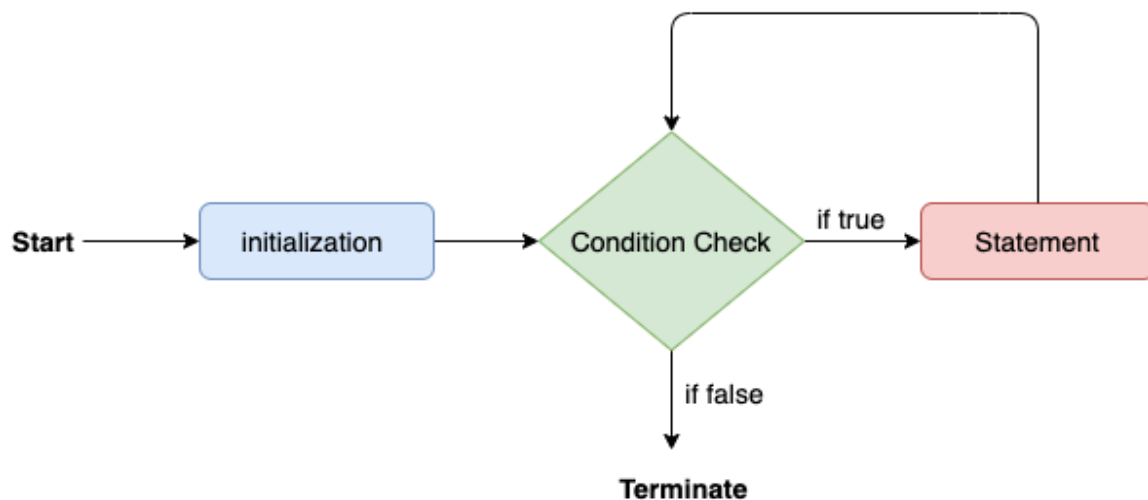
Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

1. **for**(initialization, condition, increment/decrement) {
2. //block of statements
3. }

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int sum = 0;  
        for(int j = 1; j<=10; j++) {  
            sum = sum + j;  
        }  
        System.out.println("The sum of first 10 natural numbers is " + sum);  
    }  
}
```

```
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

1. **for**(data_type var : array_name/collection_name){
2. //statements
3. }

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        String[] names = {"Java","C","C++","Python","JavaScript"};  
        System.out.println("Printing the content of the array names:\n");  
        for(String name:names) {  
            System.out.println(name);  
        }  
    }  
}
```

Output:

```
Printing the content of the array names:
```

```
Java  
C  
C++  
Python  
JavaScript
```

Java while loop

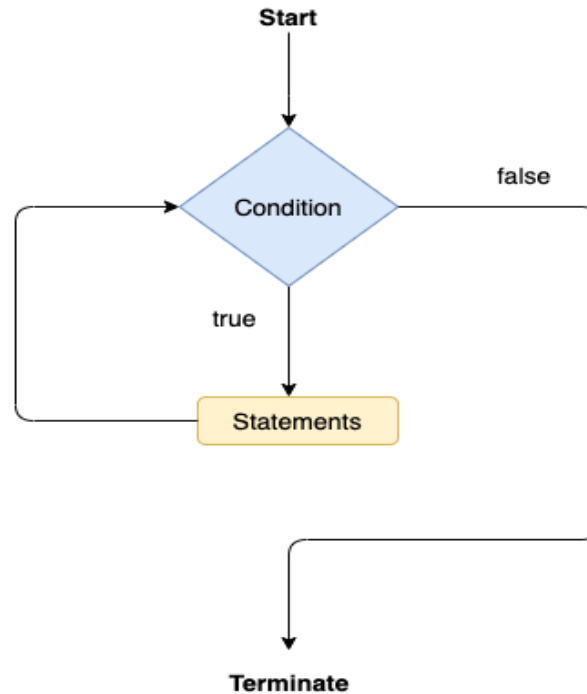
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

1. **while**(condition){
2. //looping statements
3. }

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        int i = 0;  
        System.out.println("Printing the list of first 10 even numbers \n");  
        while(i<=10) {  
            System.out.println(i);  
            i = i + 2;  
        }  
    }  
}
```

Output:

Printing the list of first 10 even numbers

0
2
4

```
6
8
10
```

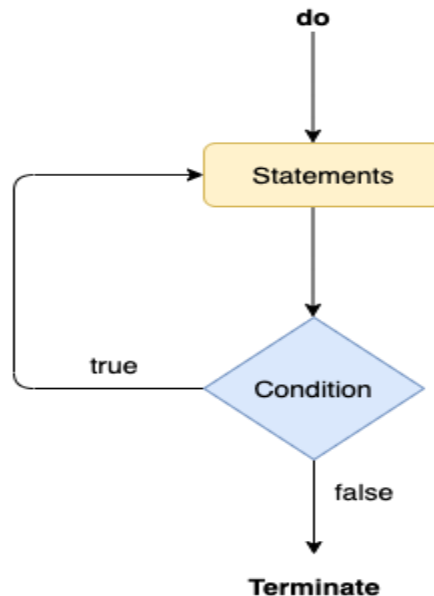
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

1. **do**
2. {
3. //statements
4. } **while** (condition);

The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int i = 0;
        System.out.println("Printing the list of first 10 even numbers \n");
        do {
            System.out.println(i);
```

```
i = i + 2;  
}while(i<=10);  
}  
}
```

Output:

Printing the list of first 10 even numbers

```
0  
2  
4  
6  
8  
10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the [break statement](#) is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

BreakExample.java

```
public class BreakExample {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        for(int i = 0; i<= 10; i++) {  
            System.out.println(i);  
            if(i==6) {  
                break;  
            }  
        }  
    }  
}
```

Output:

```
0  
1
```

```
2  
3  
4  
5  
6
```

Java continue statement

Unlike break statement, the [continue statement](#) doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        for(int i = 0; i<= 2; i++) {  
            for (int j = i; j<=5; j++) {  
                if(j == 4) {  
                    continue;  
                }  
                System.out.println(j);  
            }  
        }  
    }  
}
```

Chapter 2: Java Applets

2.1 Introduction to Java Applets

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal. There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the `java.applet.Applet` class.
- A `main()` method is not invoked on an applet, and an applet class will not define `main()`.
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

2.2 Life Cycle of an Applet

our methods in the Applet class gives you the framework on which you build any serious applet –

- **init** – This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** – This method is automatically called after the browser calls the `init` method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** – This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** – This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** – Invoked immediately after the `start()` method, and also any time the applet needs to repaint itself in the browser. The `paint()` method is actually inherited from the `java.awt`.

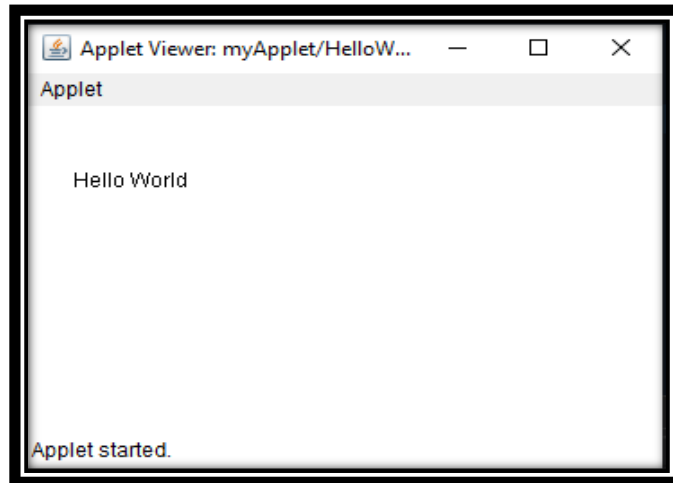
Following is a simple applet named `HelloWorldApplet.java` –

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
```

```
public void paint (Graphics g) {  
    g.drawString ("Hello World", 25, 50);  
}  
}
```

The output the applet code is depicted below.



These import statements bring the classes into the scope of our applet class –

- `java.applet.Applet`
- `java.awt.Graphics`

Without those import statements, the Java compiler would not recognize the classes `Applet` and `Graphics`, which the applet class refers to.

2.3 The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base `Applet` class provides methods that a derived `Applet` class may call to obtain information and services from the browser context.

These include methods that do the following –

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the `Applet` class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may –

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary. The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser. The <applet> tag is the basis for embedding an applet in an HTML file. Following is an example that invokes the "Hello, World" applet –

```
<html>
  <title>The Hello, World Applet</title>
  <hr>
  <applet code = "HelloWorldApplet.class" width = "320" height = "120">
    If your browser was Java-enabled, a "Hello, World"
    message would appear here.
  </applet>
  <hr>
</html>
```

Note – You can refer to [HTML Applet Tag](#) to understand more about calling applet from HTML.

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with an </applet> tag.

If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown –

```
<applet codebase = "https://amrood.com/applets" code = "HelloWorldApplet.class"
  width = "320" height = "120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example

```
<applet = "mypackage.subpackage.TestApplet.class"
  width = "320" height = "120">
```

Getting Applet Parameters

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color. The second color and the size of each square may be specified as parameters to the applet within the document.

CheckerApplet gets its parameters in the `init()` method. It may also get its parameters in the `paint()` method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.

The applet viewer or browser calls the `init()` method of each applet it runs. The viewer calls `init()` once, immediately after loading the applet. (`Applet.init()` is implemented to do nothing.) Override the default implementation to insert custom initialization code.

The `Applet.getParameter()` method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.

The following is a skeleton of `CheckerApplet.java` –

```
import java.applet.*;
import java.awt.*;
public class CheckerApplet extends Applet {
    int squareSize = 50; // initialized to default size
    public void init() {}
    private void parseSquareSize (String param) {}
    private Color parseColor (String param) {}
    public void paint (Graphics g) {}
}
```

Here are `CheckerApplet`'s `init()` and private `parseSquareSize()` methods –

```
public void init () {
    String squareSizeParam = getParameter ("squareSize");
    parseSquareSize (squareSizeParam);
    String colorParam = getParameter ("color");
```



```
Color fg = parseColor (colorParam);
setBackground (Color.black);
setForeground (fg);
}
private void parseSquareSize (String param) {
    if (param == null) return;
    try {
        squareSize = Integer.parseInt (param);
    } catch (Exception e) {
        // Let default value remain
    }
}
```

The applet calls `parseSquareSize()` to parse the `squareSize` parameter. `parseSquareSize()` calls the library method `Integer.parseInt()`, which parses a string and returns an integer. `Integer.parseInt()` throws an exception whenever its argument is invalid.

Therefore, `parseSquareSize()` catches exceptions, rather than allowing the applet to fail on bad input.

The applet calls `parseColor()` to parse the color parameter into a `Color` value. `parseColor()` does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet work.

Specifying Applet Parameters

The following is an example of an HTML file with a `CheckerApplet` embedded in it. The HTML file specifies both parameters to the applet by means of the `<param>` tag.

```
<html>
  <title>Checkerboard Applet</title>
  <hr>
  <applet code = "CheckerApplet.class" width = "480" height = "320">
    <param name = "color" value = "blue">
    <param name = "squaresize" value = "30">
  </applet>
  <hr>
</html>
```

Note – Parameter names are not case sensitive.

2.4 Event Handling in applets

Applets inherit a group of event-handling methods from the `Container` class. The `Container` class defines several methods, such as `processKeyEvent` and `processMouseEvent`, for handling particular types of events, and then one catch-all method called `processEvent`.

In order to react to an event, an applet must override the appropriate event-specific method.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;
public class ExampleEventHandling extends Applet implements MouseListener {
    StringBuffer strBuffer;
    public void init() {
        addMouseListener(this);
        strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }
    public void start() {
        addItem("starting the applet ");
    }
    public void stop() {
        addItem("stopping the applet ");
    }
    public void destroy() {
        addItem("unloading the applet");
    }
    void addItem(String word) {
        System.out.println(word);
        strBuffer.append(word);
        repaint();
    }
    public void paint(Graphics g) {
        // Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
            getWidth() - 1,
            getHeight() - 1);

        // display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20); }
    public void mouseEntered(MouseEvent event) { }
    public void mouseExited(MouseEvent event) { }
    public void mousePressed(MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
    public void mouseClicked(MouseEvent event) {
        addItem("mouse clicked! "); }
}
```

```
}
```

Now, let us call this applet as follows –

```
<html>
  <title>Event Handling</title>
  <hr>
  <applet code = "ExampleEventHandling.class"
    width = "300" height = "300">
  </applet>
  <hr>
</html>
```

Initially, the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle, "mouse clicked" will be displayed as well.

Chapter Three: Java GUI using JavaFX

3.1 Introduction

JavaFX is a Java library used to build Rich Internet Applications. The applications written using this library can run consistently across multiple platforms. The applications developed using JavaFX can run on various devices such as Desktop Computers, Mobile Phones, TVs, Tablets, etc.

To develop **GUI applications** using Java programming language, the programmers rely on libraries such as **Advanced Windowing Tool kit** and **Swing**. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.

Rich Internet Applications are those web applications which provide similar features and experience as that of desktop applications. They offer a better visual experience when compared to the normal web applications to the users. These applications are delivered as browser plug-ins or as a virtual machine and are used to transform traditional static applications into more enhanced, fluid, animated and engaging applications.

We have three main technologies using which we can develop an RIA. These include the following:

- Adobe Flash
- Microsoft Silverlight
- JavaFX

To develop GUI Applications using Java programming language, the programmers rely on libraries such as Advanced Windowing Toolkit and Swing. After the advent of JavaFX, these Java programmers can now develop GUI applications effectively with rich content.

3.2 Need for JavaFX

To develop Client Side Applications with rich features, the programmers used to depend on various libraries to add features such as Media, UI controls, Web, 2D and 3D, etc. JavaFX includes all these features in a single library. In addition to these, the developers can also access the existing features of a Java library such as Swing.

JavaFX provides a rich set of graphics and media API's and it leverages the modern Graphical Processing Unit through hardware accelerated graphics. JavaFX also provides interfaces using which developers can combine graphics animation and UI control.

One can use JavaFX with JVM based technologies such as Java, Groovy and JRuby. If developers opt for JavaFX, there is no need to learn additional technologies, as prior knowledge of any of the above-mentioned technologies will be good enough to develop RIA's using JavaFX.

3.3 Features of JavaFX

Following are some of the important features of JavaFX :

- **Written in Java** – The JavaFX library is written in Java and is available for the languages that can be executed on a JVM, which include – Java, Groovy and JRuby. These JavaFX applications are also platform independent.
- **FXML** – JavaFX features a language known as FXML, which is a HTML like declarative markup language. The sole purpose of this language is to define a user Interface.
- **Scene Builder** – JavaFX provides an application named Scene Builder. On integrating this application in IDE's such as Eclipse and NetBeans, the users can access a drag and drop design interface, which is used to develop FXML applications (just like Swing Drag & Drop and Dreamweaver Applications).
- **Swing Interoperability** – In a JavaFX application, you can embed Swing content using the Swing Node class. Similarly, you can update the existing Swing applications with JavaFX features like embedded web content and rich graphics media.
- **Built-in UI controls** – JavaFX library caters UI controls using which we can develop a full-featured application.
- **CSS like Styling** – JavaFX provides a CSS like styling. By using this, you can improve the design of your application with a simple knowledge of CSS.
- **Canvas and Printing API** – JavaFX provides Canvas, an immediate mode style of rendering API. Within the package **javafx.scene.canvas** it holds a set of classes for canvas, using which we can draw directly within an area of the JavaFX scene. JavaFX also provides classes for Printing purposes in the package **javafx.print**.
- **Rich set of API's** – JavaFX library provides a rich set of API's to develop GUI applications, 2D and 3D graphics, etc. Therefore, using this API, you can access the features of Java languages such as Generics, Annotations, Multithreading, and Lambda Expressions.
- **Integrated Graphics library** – JavaFX provides classes for 2d and 3d graphics.
- **Graphics pipeline** – JavaFX supports graphics based on the Hardware-accelerated graphics pipeline known as Prism. When used with a supported Graphic Card or GPU it offers smooth graphics. In case the system does not support graphic card then prism defaults to the software rendering stack.

3.4 JavaFX – Architecture

JavaFX provides a complete API with a rich set of classes and interfaces to build GUI applications with rich graphics. The important packages of this API are –

- **javafx.animation** – Contains classes to add transition based animations such as fill, fade, rotate, scale and translation, to the JavaFX nodes.
- **javafx.application** – Contains a set of classes responsible for the JavaFX application life cycle.
- **javafx.css** – Contains classes to add CSS-like styling to JavaFX GUI applications.
- **javafx.event** – Contains classes and interfaces to deliver and handle JavaFX events.
- **javafx.geometry** – Contains classes to define 2D objects and perform operations on them.

- **javafx.stage** – This package holds the top level container classes for JavaFX application.
- **javafx.scene** – This package provides classes and interfaces to support the scene graph. In addition, it also provides sub-packages such as canvas, chart, control, effect, image, input, layout, media, paint, shape, text, transform, web, etc. There are several components that support this rich API of JavaFX.

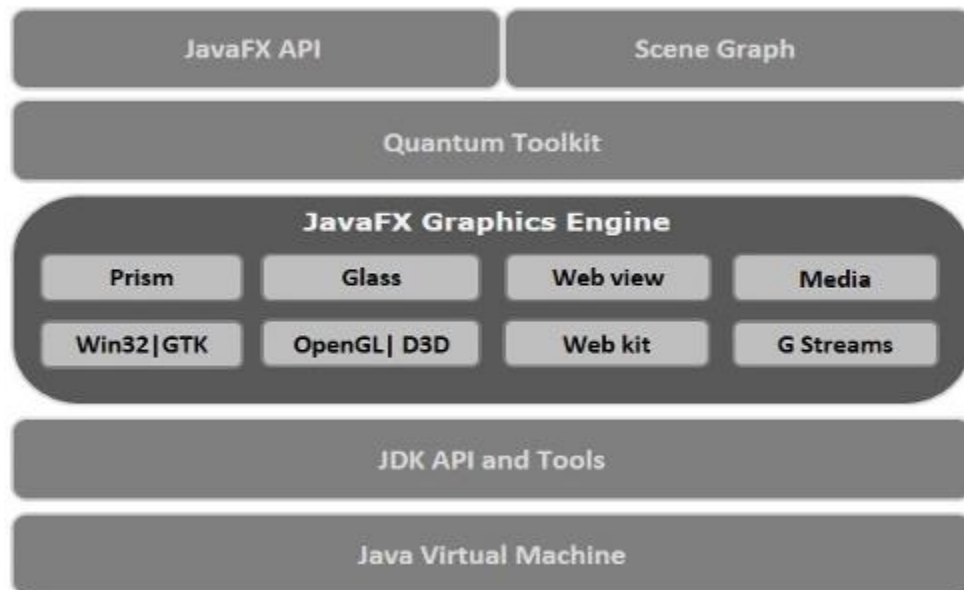


Figure 3. 1 Java FX Architecture

Scene Graph: In JavaFX, the GUI Applications were coded using a Scene Graph. A Scene Graph is the starting point of the construction of the GUI Application. It holds the (GUI) application primitives that are termed as nodes.

A node is a visual/graphical object and it may include –

- **Geometrical (Graphical) objects** – (2D and 3D) such as circle, rectangle, polygon, etc.
- **UI controls** – such as Button, Checkbox, Choice box, Text Area, etc.
- **Containers** – (layout panes) such as Border Pane, Grid Pane, Flow Pane, etc.
- **Media elements** – such as audio, video and image objects.

In general, a collection of nodes makes a scene graph. All these nodes are arranged in a hierarchical order as shown below.

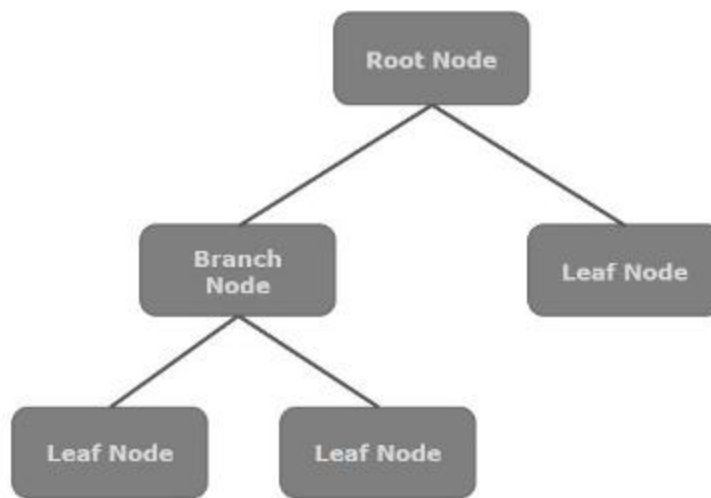


Figure 3. 2 Hierarchical order of Scene Graph

A node instance can be added to a scene graph only once. The nodes of a scene graph can have Effects, Opacity, Transforms, Event Handlers, Event Handlers, and Application Specific States.

Prism: It is a high performance hardware–accelerated graphical pipeline that is used to render the graphics in JavaFX. It can render both 2-D and 3-D graphics.

To render graphics, a Prism uses –

- DirectX 9 on Windows XP and Vista.
- DirectX 11 on Windows 7.
- OpenGL on Mac and Linux, Embedded Systems.

In case the hardware support for graphics on the system is not sufficient, then Prism uses the software render path to process the graphics.

GWT (Glass Windowing Toolkit): As the name suggests, GWT provides services to manage Windows, Timers, Surfaces and Event Queues. GWT connects the JavaFX Platform to the Native Operating System.

Quantum Toolkit: It is an abstraction over the low-level components of Prism, Glass, Media Engine, and Web Engine. It ties Prism and GWT together and makes them available to JavaFX.

WebView: Using JavaFX, you can also embed HTML content in to a scene graph. WebView is the component of JavaFX which is used to process this content. It uses a technology called **Web Kit**, which is an internal open-source web browser engine. This component supports different web technologies like HTML5, CSS, JavaScript, DOM and SVG.

Using WebView, you can –

- Render HTML content from local or remote URL.
- Support history and provide Back and Forward navigation.
- Reload the content.

- Apply effects to the web component.
- Edit the HTML content.
- Execute JavaScript commands.
- Handle events.

In general, using `WebView`, you can control web content from Java.

Media Engine: The JavaFX media engine is based on an open-source engine known as a Streamer. This media engine supports the playback of video and audio content.

The package **`javafx.scene.media`** contains the classes and interfaces to provide media functionality in JavaFX. It is provided in the form of three components, which are –

- **Media Object** – This represents a media file
- **Media Player** – To play media content.
- **Media View** – To display media.

3.5 JavaFX Application Structure

In general, a JavaFX application will have three major components namely **Stage**, **Scene** and **Nodes**.

Stage: A stage (a window) contains all the objects of a JavaFX application. It is represented by `Stage` class of the package **`javafx.stage`**. The primary stage is created by the platform itself. The created stage object is passed as an argument to the **`start()`** method of the `Application` class. A stage has two parameters determining its position namely Width and Height. It is divided as Content Area and Decorations (Title Bar and Borders). There are five types of stages available :

- Decorated
- Undecorated
- Transparent
- Unified
- Utility

You have to call the **`show()`** method to display the contents of a stage.

Scene: A scene represents the physical contents of a JavaFX application. It contains all the contents of a scene graph. The class `Scene` of the package **`javafx.scene`** represents the scene object. At an instance, the scene object is added to only one stage. You can create a scene by instantiating the `Scene` Class. You can opt for the size of the scene by passing its dimensions (height and width) along with the **root node** to its constructor.

Scene Graph and Nodes: A **scene graph** is a tree-like data structure (hierarchical) representing the contents of a scene. In contrast, a **node** is a visual/graphical object of a scene graph.

A node may include –

- Geometrical (Graphical) objects (2D and 3D) such as – Circle, Rectangle, Polygon, etc.
- UI Controls such as – Button, Checkbox, Choice Box, Text Area, etc.
- Containers (Layout Panes) such as Border Pane, Grid Pane, Flow Pane, etc.
- Media elements such as Audio, Video and Image Objects.

The **Node** Class of the package **`javafx.scene`** represents a node in JavaFX, this class is the super class of all the nodes.

As discussed earlier a node is of three types:

- **Root Node** : the first Scene Graph is known as the Root node.
- **Branch Node/Parent Node**: The node with child nodes are known as branch/parent nodes. The abstract class named **Parent** of the package **javafx.scene** is the base class of all the parent nodes, and those parent nodes will be of the following types –
 - **Group** – A group node is a collective node that contains a list of children nodes. Whenever the group node is rendered, all its child nodes are rendered in order. Any transformation, effect state applied on the group will be applied to all the child nodes.
 - **Region** – It is the base class of all the JavaFX Node based UI Controls, such as Chart, Pane and Control.
 - **WebView** – This node manages the web engine and displays its contents.
- **Leaf Node** – The node without child nodes is known as the leaf node. For example, Rectangle, Ellipse, Box, ImageView, MediaView are examples of leaf nodes.

It is mandatory to pass the root node to the scene graph. If the Group is passed as root, all the nodes will be clipped to the scene and any alteration in the size of the scene will not affect the layout of the scene.

3.6 Creating a JavaFX Application

To create a JavaFX application, you need to instantiate the Application class and implement its abstract method **start()**.

The **Application** class of the package **javafx.application** is the entry point of the application in JavaFX. To create a JavaFX application, you need to inherit this class and implement its abstract method **start()**. In this method, you need to write the entire code for the JavaFX graphics

In the **main** method, you have to launch the application using the **launch()** method. This method internally calls the **start()** method of the Application class as shown in the following program.

```
public class JavafxSample extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
    }
    public static void main(String args[]){
        launch(args);
    }
}
```

Within the **start()** method, in order to create a typical JavaFX application, you need to follow the steps given below –

- Prepare a scene graph with the required nodes.
- Prepare a Scene with the required dimensions and add the scene graph (root node of the scene graph) to it.
- Prepare a stage and add the scene to the stage and display the contents of the stage.

As per your application, you need to prepare a **scene graph** with required nodes. Since the root node is the first node, you need to create a **root node**. As a root node, you can choose from the **Group, Region or WebView**.

Group – A Group node is represented by the class named **Group** which belongs to the package **javafx.scene**, you can create a Group node by instantiating this class as shown below.

```
Group root = new Group();
```

The **getChildren()** method of the **Group** class gives you an object of the **ObservableList** class which holds the nodes. We can retrieve this object and add nodes to it as shown below.

```
//Retrieving the observable list object
```

```
ObservableList list = root.getChildren();
```

```
//Setting the text object as a node
```

```
list.add(NodeObject);
```

We can also add Node objects to the group, just by passing them to the **Group** class and to its constructor at the time of instantiation, as shown below.

```
Group root = new Group(NodeObject);
```

Region – It is the Base class of all the JavaFX Node-based UI Controls, such as –

- **Chart** – This class is the base class of all the charts and it belongs to the package **javafx.scene.chart** and sub classes **pieChart** and **XYChart**
- **Pane** – A Pane is the base class of all the layout panes such as **AnchorPane, BorderPane, DialogPane**, etc. This class belong to a package that is called as – **javafx.scene.layout**. You can use these classes to insert predefined layouts in your application.
- **Control** – It is the base class of the User Interface controls such as **Accordion, ButtonBar, ChoiceBox, ComboBoxBase, HTML editor**, etc. This class belongs to the package **javafx.scene.control**.

You can use these classes to insert various UI elements in your application.

In a Group, you can instantiate any of the above-mentioned classes and use them as root nodes, as shown in the following program.

```
//Creating a Stack Pane
```

```
StackPane pane = new StackPane();
```

```
//Adding text area to the pane
```

```
ObservableList list = pane.getChildren();
```

```
list.add(NodeObject);
```

A JavaFX scene is represented by the **Scene** class of the package **javafx.scene**. You can create a Scene by instantiating this class as shown in the following code block.

While instantiating, it is mandatory to pass the root object to the constructor of the scene class.

```
Scene scene = new Scene(root);
```

You can also pass two parameters of double type representing the height and width of the scene as shown below.

```
Scene scene = new Scene(root, 600, 300);
```

This is the container of any JavaFX application and it provides a window for the application. It is represented by the **Stage** class of the package **javafx.stage**. An object of this class is passed as a parameter of the **start()** method of the **Application** class.

Using this object, you can perform various operations on the stage. Primarily you can perform the following

- Set the title for the stage using the method **setTitle()**.
- Attach the scene object to the stage using the **setScene()** method.
- Display the contents of the scene using the **show()** method as shown below.

```
//Setting the title to Stage.
```

```
primaryStage.setTitle("Sample application");
```

```
//Setting the scene to Stage
```

```
primaryStage.setScene(scene);
```

```
//Displaying the stage
```

```
primaryStage.show();
```

The following program generates an empty JavaFX window. Save this code in a file with the name **JavafxSample.java**

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
public class JavafxSample extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        //creating a Group object
        Group group = new Group();
        //Creating a Scene by passing the group object, height and width
        Scene scene = new Scene(group ,600, 300);
        //setting color to the scene
        scene.setFill(Color.BROWN);
    }
}
```

```
//Setting the title to Stage.  
primaryStage.setTitle("Sample Application");  
//Adding the scene to Stage  
primaryStage.setScene(scene);  
//Displaying the contents of the stage  
primaryStage.show();  
}  
public static void main(String args[]){  
    launch(args);  
}  
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac JavafxSample.java
```

```
java JavafxSample
```

On executing, the above program generates a JavaFX window as shown below.



3.7 Lifecycle of JavaFX Application

The JavaFX Application class has three life cycle methods, which are –

- **start()** – The entry point method where the JavaFX graphics code is to be written.
- **stop()** – An empty method which can be overridden, here you can write the logic to stop the application.
- **init()** – An empty method which can be overridden, but you cannot create stage or scene in this method.

In addition to these, it provides a static method named **launch()** to launch JavaFX application.

Since the **launch()** method is static, you need to call it from a static context (main generally). Whenever a JavaFX application is launched, the following actions will be carried out (in the same order).

- An instance of the application class is created.
- **Init()** method is called.
- The **start()** method is called.
- The launcher waits for the application to finish and calls the **stop()** method.

3.8 Terminating the JavaFX Application

When the last window of the application is closed, the JavaFX application is terminated implicitly. You can turn this behavior off by passing the Boolean value “False” to the static method **setImplicitExit()** (should be called from a static context). You can terminate a JavaFX application explicitly using the methods **Platform.exit()** or **System.exit(int)**.

3.9 JAVAFX layout components

The arrangement of the components within the container is called the Layout of the container. We can also say that we followed a layout as it includes placing all the components at a particular position within the container.

JavaFX provides several predefined layouts such as **HBox**, **VBox**, **Border Pane**, **Stack Pane**, **Text Flow**, **Anchor Pane**, **Title Pane**, **Grid Pane**, **Flow Panel**, etc.

Each of the above mentioned layout is represented by a class and all these classes belongs to the package **javafx.layout**. The class named **Pane** is the base class of all the layouts in JavaFX.

To create a layout, you need to –

- Create node.
- Instantiate the respective class of the required layout.
- Set the properties of the layout.
- Add all the created nodes to the layout.

First of all, create the required nodes of the JavaFX application by instantiating their respective classes. For example, if you want to have a text field and two buttons namely, play and stop in a HBox layout - you will have to initially create those nodes as shown in the following code block:

```
//Creating a text field
```

```
TextField textField = new TextField();
```

```
//Creating the play button
```

```
Button playButton = new Button("Play");
```

```
//Creating the stop button
```

```
Button stopButton = new Button("stop");
```

After creating the nodes (and completing all the operations on them), instantiate the class of the required layout. For Example, if you want to create a Hbox layout, you need to instantiate this class as follows.

```
HBox hbox = new HBox();
```

After instantiating the class, you need to set the properties of the layout using their respective setter methods. For example – If you want to set space between the created nodes in the HBox layout, then you need to set value to the property named spacing. This can be done by using the setter method **setSpacing()** as shown below –

```
hbox.setSpacing(10);
```

Finally, you need to add the object of the shape to the group by passing it as a parameter of the constructor as shown below.

```
//Creating a Group object
```

```
Group root = new Group(line);
```

Following are the various Layout panes (classes) provided by JavaFX. These classes exist in the package **javafx.scene.layout**.

S.No	Shape & Description
1	HBox The HBox layout arranges all the nodes in our application in a single horizontal row. The class named HBox of the package javafx.scene.layout represents the text horizontal box layout.
2	VBox The VBox layout arranges all the nodes in our application in a single vertical column. The class named VBox of the package javafx.scene.layout represents the text Vertical box layout.
3	BorderPane The Border Pane layout arranges the nodes in our application in top, left, right, bottom and center positions. The class named BorderPane of the package javafx.scene.layout represents the border pane layout.
4	StackPane The stack pane layout arranges the nodes in our application on top of another just like in a stack. The node added first is placed at the bottom of the stack and the next node is placed on top of it. The class named StackPane of the package javafx.scene.layout represents the stack pane layout.

5	TextFlow The Text Flow layout arranges multiple text nodes in a single flow. The class named TextFlow of the package javafx.scene.layout represents the text flow layout.
6	AnchorPane The Anchor pane layout anchors the nodes in our application at a particular distance from the pane. The class named AnchorPane of the package javafx.scene.layout represents the Anchor Pane layout.
7	TilePane The Tile Pane layout adds all the nodes of our application in the form of uniformly sized tiles. The class named TilePane of the package javafx.scene.layout represents the TilePane layout.
8	GridPane The Grid Pane layout arranges the nodes in our application as a grid of rows and columns. This layout comes handy while creating forms using JavaFX. The class named GridPane of the package javafx.scene.layout represents the GridPane layout.
9	FlowPane The flow pane layout wraps all the nodes in a flow. A horizontal flow pane wraps the elements of the pane at its height, while a vertical flow pane wraps the elements at its width. The class named FlowPane of the package javafx.scene.layout represents the Flow Pane layout.

3.10 JavaFX - UI Controls

Every user interface considers the following three main aspects –

- **UI elements** – These are the core visual elements which the user eventually sees and interacts with. JavaFX provides a huge list of widely used and common elements varying from basic to complex, which we will cover in this tutorial.

- **Layouts** – They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface).
- **Behavior** – These are events which occur when the user interacts with UI elements. This part will be covered in the Event Handling chapter.

JavaFX provides several classes in the package **javafx.scene.control**. To create various GUI components (controls), JavaFX supports several controls such as date picker, button text field, etc.

Each control is represented by a class; you can create a control by instantiating its respective class.

Following is the list of commonly used controls while the GUI is designed using JavaFX.

S.No	Control & Description
1	Label A Label object is a component for placing text.
2	Button This class creates a labeled button.
3	ColorPicker A ColorPicker provides a pane of controls designed to allow a user to manipulate and select a color.
4	CheckBox A CheckBox is a graphical component that can be in either an on(true) or off (false) state.
5	RadioButton The RadioButton class is a graphical component, which can either be in a ON (true) or OFF (false) state in a group.
6	ListView A ListView component presents the user with a scrolling list of text items.
7	TextField A TextField object is a text component that allows for the editing of a single line of text.
8	PasswordField

	A PasswordField object is a text component specialized for password entry.
9	Scrollbar A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
10	FileChooser A FileChooser control represents a dialog window from which the user can select a file.
11	ProgressBar As the task progresses towards completion, the progress bar displays the task's percentage of completion.
12	Slider A Slider lets the user graphically select a value by sliding a knob within a bounded interval.

The following program is an example which displays a login page in JavaFX. Here, we are using the controls **label**, **text field**, **password field** and **button**.

Save this code in a file with the name **LoginPage.java**.

```
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.geometry.Insets;
import javafx.geometry.Pos;

import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.PasswordField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.scene.control.TextField;
import javafx.stage.Stage;

public class LoginPage extends Application {
    @Override
    public void start(Stage stage) {
        //creating label email
```

```
Text text1 = new Text("Email");

//creating label password
Text text2 = new Text("Password");

//Creating Text Filed for email
TextField textField1 = new TextField();

//Creating Text Filed for password
PasswordField textField2 = new PasswordField();

//Creating Buttons
Button button1 = new Button("Submit");
Button button2 = new Button("Clear");

//Creating a Grid Pane
GridPane gridPane = new GridPane();

//Setting size for the pane
gridPane.setMinSize(400, 200);

//Setting the padding
gridPane.setPadding(new Insets(10, 10, 10, 10));

//Setting the vertical and horizontal gaps between the columns
gridPane.setVgap(5);
gridPane.setHgap(5);

//Setting the Grid alignment
gridPane.setAlignment(Pos.CENTER);

//Arranging all the nodes in the grid
gridPane.add(text1, 0, 0);
gridPane.add(textField1, 1, 0);
gridPane.add(text2, 0, 1);
gridPane.add(textField2, 1, 1);
gridPane.add(button1, 0, 2);
gridPane.add(button2, 1, 2);

//Styling nodes
```

```
button1.setStyle("-fx-background-color: darkslateblue; -fx-text-fill: white;");
button2.setStyle("-fx-background-color: darkslateblue; -fx-text-fill: white;");

text1.setStyle("-fx-font: normal bold 20px 'serif' ");
text2.setStyle("-fx-font: normal bold 20px 'serif' ");
gridPane.setStyle("-fx-background-color: BEIGE;");

//Creating a scene object
Scene scene = new Scene(gridPane);

//Setting title to the Stage
stage.setTitle("CSS Example");

//Adding scene to the stage
stage.setScene(scene);

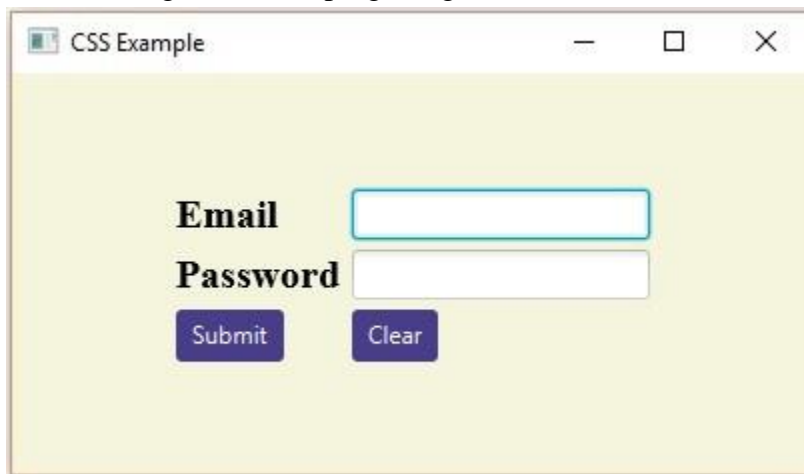
//Displaying the contents of the stage
stage.show();
}
public static void main(String args[]){
    launch(args);
}
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac LoginPage.java
```

```
java LoginPage
```

On executing, the above program generates a JavaFX window as shown below.



The following program is an example of a registration form, which demonstrates controls in JavaFX such as **Date Picker, Radio Button, Toggle Button, Check Box, List View, Choice List**, etc.

Save this code in a file with the name **Registration.java**.

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

import javafx.geometry.Insets;
import javafx.geometry.Pos;

import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.DatePicker;
import javafx.scene.control.ListView;
import javafx.scene.control.RadioButton;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.scene.control.TextField;
import javafx.scene.control.ToggleGroup;
import javafx.scene.control.ToggleButton;
import javafx.stage.Stage;

public class Registration extends Application {
    @Override
    public void start(Stage stage) {
        //Label for name
        Text nameLabel = new Text("Name");

        //Text field for name
        TextField nameText = new TextField();

        //Label for date of birth
        Text dobLabel = new Text("Date of birth");

        //date picker to choose date
        DatePicker datePicker = new DatePicker();
```

```
//Label for gender
Text genderLabel = new Text("gender");

//Toggle group of radio buttons
ToggleGroup groupGender = new ToggleGroup();
RadioButton maleRadio = new RadioButton("male");
maleRadio.setToggleGroup(groupGender);
RadioButton femaleRadio = new RadioButton("female");
femaleRadio.setToggleGroup(groupGender);

//Label for reservation
Text reservationLabel = new Text("Reservation");

//Toggle button for reservation
ToggleButton Reservation = new ToggleButton();
ToggleButton yes = new ToggleButton("Yes");
ToggleButton no = new ToggleButton("No");
ToggleGroup groupReservation = new ToggleGroup();
yes.setToggleGroup(groupReservation);
no.setToggleGroup(groupReservation);

//Label for technologies known
Text technologiesLabel = new Text("Technologies Known");

//check box for education
CheckBox javaCheckBox = new CheckBox("Java");
javaCheckBox.setIndeterminate(false);

//check box for education
CheckBox dotnetCheckBox = new CheckBox("DotNet");
javaCheckBox.setIndeterminate(false);

//Label for education
Text educationLabel = new Text("Educational qualification");

//list View for educational qualification
ObservableList<String> names = FXCollections.observableArrayList(
    "Engineering", "MCA", "MBA", "Graduation", "MTECH", "Mphil", "Phd");
ListView<String> educationListView = new ListView<String>(names);
```

```
//Label for location
Text locationLabel = new Text("location");

//Choice box for location
ChoiceBox locationchoiceBox = new ChoiceBox();
locationchoiceBox.getItems().addAll
    ("Hyderabad", "Chennai", "Delhi", "Mumbai", "Vishakhapatnam");

//Label for register
Button buttonRegister = new Button("Register");

//Creating a Grid Pane
GridPane gridPane = new GridPane();

//Setting size for the pane
gridPane.setMinSize(500, 500);

//Setting the padding
gridPane.setPadding(new Insets(10, 10, 10, 10));

//Setting the vertical and horizontal gaps between the columns
gridPane.setVgap(5);
gridPane.setHgap(5);

//Setting the Grid alignment
gridPane.setAlignment(Pos.CENTER);

//Arranging all the nodes in the grid
gridPane.add(nameLabel, 0, 0);
gridPane.add(nameText, 1, 0);

gridPane.add(dobLabel, 0, 1);
gridPane.add(datePicker, 1, 1);

gridPane.add(genderLabel, 0, 2);
gridPane.add(maleRadio, 1, 2);
gridPane.add(femaleRadio, 2, 2);
gridPane.add(reservationLabel, 0, 3);
gridPane.add(yes, 1, 3);
```

```
gridPane.add(no, 2, 3);

gridPane.add(technologiesLabel, 0, 4);
gridPane.add(javaCheckBox, 1, 4);
gridPane.add(dotnetCheckBox, 2, 4);

gridPane.add(educationLabel, 0, 5);
gridPane.add(educationListView, 1, 5);

gridPane.add(locationLabel, 0, 6);
gridPane.add(locationchoiceBox, 1, 6);

gridPane.add(buttonRegister, 2, 8);

//Styling nodes
buttonRegister.setStyle(
    "-fx-background-color: darkslateblue; -fx-textfill: white;");

nameLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
dobLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
genderLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
reservationLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
technologiesLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
educationLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
locationLabel.setStyle("-fx-font: normal bold 15px 'serif' ");
//Setting the back ground color
gridPane.setStyle("-fx-background-color: BEIGE;");
//Creating a scene object
Scene scene = new Scene(gridPane);
//Setting title to the Stage
stage.setTitle("Registration Form");
//Adding scene to the stage
stage.setScene(scene);
//Displaying the contents of the stage
stage.show();
}
public static void main(String args[]){
    launch(args);
}
```

```
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac Registration.java
```

```
java Registration
```

On executing, the above program generates a JavaFX window as shown below.

The screenshot shows a JavaFX window titled "Registration Form". The form contains the following fields and controls:

- Name:** A text input field.
- Date of birth:** A date picker field.
- gender:** Two radio buttons labeled "male" and "female".
- Reservation:** Two buttons labeled "Yes" and "No".
- Technologies Known:** Two checkboxes labeled "Java" and "DotNet".
- Educational qualification:** A list box containing the following items: Engineering, MCA, MBA, Graduation, MTECH, Mphil, and Phd.
- location:** A dropdown menu with a list of cities: Hyderabad, Chennai, Delhi, Mumbai, and Vishakhapatnam.
- Register:** A blue button located at the bottom right of the form.

3.11 JavaFX Event Handling

In JavaFX, we can develop GUI applications, web applications and graphical applications. In such applications, whenever a user interacts with the application (nodes), an event is said to have been occurred. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Event Handling is the mechanism that controls the event and decides what should happen, if an event occurs. This mechanism has the code which is known as an event handler that is executed when an event occurs.

JavaFX provides handlers and filters to handle events. In JavaFX every event has –

- **Target** – The node on which an event occurred. A target can be a window, scene, and a node.
- **Source** – The source from which the event is generated will be the source of the event. In the above scenario, mouse is the source of the event.
- **Type** – Type of the occurred event; in case of mouse event – mouse pressed, mouse released are the type of events.

3.11.1 Types of Events

The events can be broadly classified into the following two categories –

- **Foreground Events** – Those events which require the direct interaction of a user. They are generated as consequences of a person interacting with the graphical components in a Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page, etc.
- **Background Events** – Those events that don't require the interaction of end-user are known as background events. The operating system interruptions, hardware or software failure, timer expiry, operation completion are the example of background events.

3.11.2 Events in JavaFX

JavaFX provides support to handle a wide varieties of events. The class named **Event** of the package **javafx.event** is the base class for an event.

An instance of any of its subclass is an event. JavaFX provides a wide variety of events. Some of them are listed below.

- **Mouse Event** – This is an input event that occurs when a mouse is clicked. It is represented by the class named **MouseEvent**. It includes actions like mouse clicked, mouse pressed, mouse released, mouse moved, mouse entered target, mouse exited target, etc.
- **Key Event** – This is an input event that indicates the key stroke occurred on a node. It is represented by the class named **KeyEvent**. This event includes actions like key pressed, key released and key typed.
- **Drag Event** – This is an input event which occurs when the mouse is dragged. It is represented by the class named **DragEvent**. It includes actions like drag entered, drag dropped, drag entered target, drag exited target, drag over, etc.
- **Window Event** – This is an event related to window showing/hiding actions. It is represented by the class named **WindowEvent**. It includes actions like window hiding, window shown, window hidden, window showing, etc.

```
//Creating the mouse event handler
EventHandler<MouseEvent> eventHandler = new EventHandler<MouseEvent>() {
    @Override
```

```
public void handle(MouseEvent e) {  
    System.out.println("Hello World");  
    circle.setFill(Color.DARKSLATEBLUE);  
}  
};  
//Adding event Filter  
Circle.addEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);
```

In the same way, you can remove a filter using the method `removeEventFilter()` as shown below
`circle.removeEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);`

Following is an example demonstrating the event handling in JavaFX using the event filters. Save this code in a file with name **EventFiltersExample.java**.

```
import javafx.application.Application;  
import static javafx.application.Application.launch;  
import javafx.event.EventHandler;  
import javafx.scene.Group;  
import javafx.scene.Scene;  
import javafx.scene.input.MouseEvent;  
import javafx.scene.paint.Color;  
import javafx.scene.shape.Circle;  
import javafx.scene.text.Font;  
import javafx.scene.text.FontWeight;  
import javafx.scene.text.Text;  
import javafx.stage.Stage;  
  
public class EventFiltersExample extends Application {  
    @Override  
    public void start(Stage stage) {  
        //Drawing a Circle  
        Circle circle = new Circle();  
        //Setting the position of the circle  
        circle.setCenterX(300.0f);  
        circle.setCenterY(135.0f);  
        //Setting the radius of the circle  
        circle.setRadius(25.0f);  
        //Setting the color of the circle  
        circle.setFill(Color.BROWN);  
        //Setting the stroke width of the circle  
        circle.setStrokeWidth(20);
```

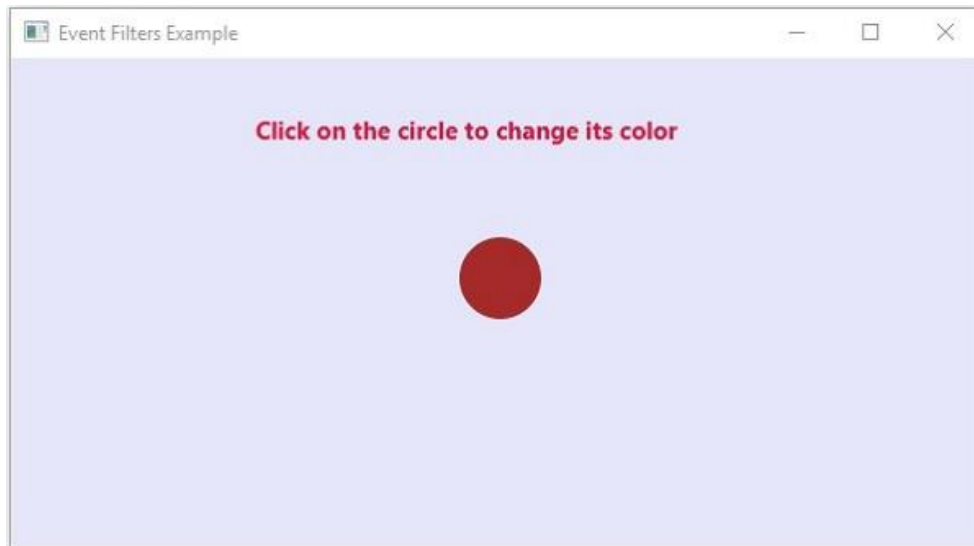
```
//Setting the text
Text text = new Text("Click on the circle to change its color");
//Setting the font of the text
text.setFont(Font.font(null, FontWeight.BOLD, 15));
//Setting the color of the text
text.setFill(Color.CRIMSON);
//setting the position of the text
text.setX(150);
text.setY(50);
//Creating the mouse event handler
EventHandler<MouseEvent> eventHandler = new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent e) {
        System.out.println("Hello World");
        circle.setFill(Color.DARKSLATEBLUE);
    }
};
//Registering the event filter
circle.addEventFilter(MouseEvent.MOUSE_CLICKED, eventHandler);
//Creating a Group object
Group root = new Group(circle, text);
//Creating a scene object
Scene scene = new Scene(root, 600, 300);
//Setting the fill color to the scene
scene.setFill(Color.LAVENDER);
//Setting title to the Stage
stage.setTitle("Event Filters Example");
//Adding scene to the stage
stage.setScene(scene);

//Displaying the contents of the stage
stage.show();
}
public static void main(String args[]){
    launch(args);
}
}
```

Compile and execute the saved java file from the command prompt using the following commands.

```
javac EventFiltersExample.java  
java EventFiltersExample
```

On executing, the above program generates a JavaFX window as shown below.



3.12 JavaFX Shapes

In some of the applications, we need to show two dimensional shapes to the user. However, JavaFX provides the flexibility to create our own 2D shapes on the screen. There are various classes which can be used to implement 2D shapes and 3D in our application. All these classes reside in **javafx.scene.shape** package. This package contains the classes which represent different types of 2D shapes. There are several methods in the classes which deal with the coordinates regarding 2D shape creation.

In general, a two dimensional shape can be defined as the geometrical figure that can be drawn on the coordinate system consisting of X and Y planes. However, this is different from 3D shapes in the sense that each point of the 2D shape always consists of two coordinates (X,Y). Using JavaFX, we can create 2D shapes such as Line, Rectangle, Circle, Ellipse, Polygon, Cubic Curve, quad curve, Arc, etc. The class **javafx.scene.shape.Shape** is the base class for all the shape classes.

How to create 2D shapes?

As we have mentioned earlier that every shape is represented by a specific class of the package **javafx.scene.shape**. For creating a two dimensional shape, the following instructions need to be followed.

1. Instantiate the respective class : for example, **Rectangle rect = new Rectangle()**
2. Set the required properties for the class using instance setter methods: for example,
 - `rect.setX(10);`
 - `rect.setY(20);`
 - `rect.setWidth(100);`
 - `rect.setHeight(100);`

3. Add class object to the Group layout: for example,

- Group root = **new** Group();
- root.getChildren().add(rect);

The following table consists of the JavaFX shape classes along with their descriptions.

Shape	Description
Line	In general, Line is the geometrical figure which joins two (X,Y) points on 2D coordinate system. In JavaFX, javafx.scene.shape.Line class needs to be instantiated in order to create lines.
Rectangle	In general, Rectangle is the geometrical figure with two pairs of two equal sides and four right angles at their joint. In JavaFX, javafx.scene.shape.Rectangle class needs to be instantiated in order to create Rectangles.
Ellipse	In general, ellipse can be defined as a curve with two focal points. The sum of the distances to the focal points are constant from each point of the ellipse. In JavaFX, javafx.scene.shape.Ellipse class needs to be instantiated in order to create Ellipse.
Arc	Arc can be defined as the part of the circumference of the circle or ellipse. In JavaFX, javafx.scene.shape.Arc class needs to be instantiated in order to create Arcs.
Circle	A circle is the special type of Ellipse having both the focal points at the same location. In JavaFX, Circle can be created by instantiating javafx.scene.shape.Circle class.
Polygon	Polygon is a geometrical figure that can be created by joining the multiple Co-planar line segments. In JavaFX, javafx.scene.shape.Polygon class needs to be instantiated in order to create polygon.
Cubic Curve	A Cubic curve is a curve of degree 3 in the XY plane. In JavaFX, javafx.scene.shape.CubicCurve class needs to be instantiated in order to create Cubic Curves.
Quad Curve	A Quad Curve is a curve of degree 2 in the XY plane. In JavaFX, javafx.scene.shape.QuadCurve class needs to create QuadCurve.

3.12.1 JavaFX Shape Properties

All the JavaFX 2D shape classes acquire the common properties defined by **JavaFX.scene.shape.Shape** class. In the following table, we have described the common shape properties.

Property	Description	Setter Methods
----------	-------------	----------------

Fill	Used to fill the shape with a defined paint. This is a object <paint> type property.	setFill(Paint)
smooth	This is a boolean type property. If true is passes then the edges of the shape will become smooth.	setSmooth(boolean)
strokeDashOffset	It defines the distances in the coordinate system which shows the shapes in the dashing patterns. This is a double type property.	setStrokeDashOffset(Double)
strokeLineCap	It represents the style of the line end cap. It is a strokeLineCap type property.	setStrokeLineCap(StrokeLineCap)
strokeLineJoin	It represents the style of the joint of the two paths.	setStrokeLineJoin(StrokeLineJoin)
strokeMiterLimit	It applies the limitation on the distance between the inside and outside points of a joint. It is a double type property.	setStrokeMiterLimit(Double)
Stroke	It is a colour type property which represents the colour of the boundary line of the shape.	setStroke(paint)
strokeType	It represents the type of the stroke (where the boundary line will be imposed to the shape) whether inside, outside or centred.	setStrokeType(StrokeType)
strokeWidth	It represents the width of the stroke.	setStrokeWidth(Double)

3.12.2 JavaFX Line

In general, Line can be defined as the geometrical structure which joins two points (X1,Y1) and (X2,Y2) in a X-Y coordinate plane. JavaFX allows the developers to create the line on the GUI of a JavaFX application. JavaFX library provides the class **Line** which is the part of **javafx.scene.shape** package.

How to create a Line?

Follow the following instructions to create a Line.

- Instantiate the class **javafx.scene.shape.Line**.
- set the required properties of the class object.

- Add class object to the group

Line class contains various properties described below.

Property	Description	Setter Methods
endX	The X coordinate of the end point of the line	setEndX(Double)
endY	The y coordinate of the end point of the line	setEndY(Double)
startX	The x coordinate of the starting point of the line	setStartX(Double)
startY	The y coordinate of the starting point of the line	setStartY(Double)

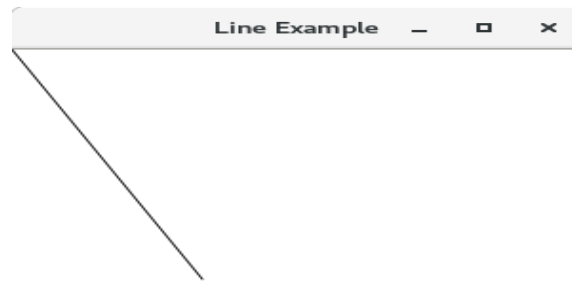
Example 1:

```

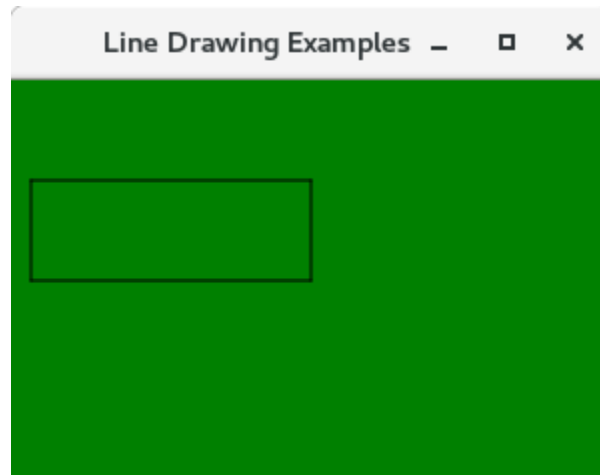
package application;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.shape.Line;
import javafx.stage.Stage;
public class LineDrawingExamples extends Application{
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        Line line = new Line(); //instantiating Line class
        line.setStartX(0); //setting starting X point of Line
        line.setStartY(0); //setting starting Y point of Line
        line.setEndX(100); //setting ending X point of Line
        line.setEndY(200); //setting ending Y point of Line
        Group root = new Group(); //Creating a Group
        root.getChildren().add(line); //adding the class object //to the group
        Scene scene = new Scene(root,300,300);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Line Example");
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}

```

```
}
```

Output:**Example 2 : Creating Multiple Lines**

```
package application;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.stage.Stage;
public class LineDrawingExamples extends Application{
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        primaryStage.setTitle("Line Drawing Examples");
        Line line1 = new Line(10,50,150,50); //Line(startX,startY,endX,endY)
        Line line2 = new Line(10,100,150,100);
        Line line3 = new Line(10,50,10,100);
        Line line4 = new Line(150,50,150,100);
        Group root = new Group();
        root.getChildren().addAll(line1,line2,line3,line4);
        Scene scene = new Scene (root,300,200,Color.GREEN);
        primaryStage.setScene(scene);
        primaryStage.show();
    } }
```

3.12.3 JavaFX Rectangle

In general, Rectangles can be defined as the geometrical figure consists of four sides, out of which, the opposite sides are always equal and the angle between the two adjacent sides is 90 degree. A Rectangle with four equal sides is called square. JavaFX library allows the developers to create a rectangle by instantiating **javafx.scene.shape.Rectangle** class.

Properties

Property	Description	Setter Method
ArcHeight	Vertical diameter of the arc at the four corners of rectangle	setArcHeight(Double height)
ArcWidth	Horizontal diameter of the arc at the four corners of the rectangle	setArcWidth(Double Width)
Height	Defines the height of the rectangle	setHeight(Double height)
Width	Defines the width of the rectangle	setWidth(Double width)
X	X coordinate of the upper left corner	setX(Double X-value)
Y	Y coordinate of the upper left corner	setY(Double(Y-value)

Example 1:

package application;

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
public class Shape_Example extends Application{
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        primaryStage.setTitle("Rectangle Example");
        Group group = new Group(); //creating Group
        Rectangle rect=new Rectangle(); //instantiating Rectangle
        rect.setX(20); //setting the X coordinate of upper left //corner of rectangle
        rect.setY(20); //setting the Y coordinate of upper left //corner of rectangle
        rect.setWidth(100); //setting the width of rectangle
        rect.setHeight(100); // setting the height of rectangle
        group.getChildren().addAll(rect); //adding rectangle to the //group
        Scene scene = new Scene(group,200,300,Color.GRAY);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Output:



3.12.4 JavaFX Circle

A circle is a special type of ellipse with both of the focal points at the same position. Its horizontal radius is equal to its vertical radius. JavaFX allows us to create Circle on the GUI of any application by just instantiating **javafx.scene.shape.Circle** class. Just set the class properties by using the instance setter methods and add the class object to the Group.

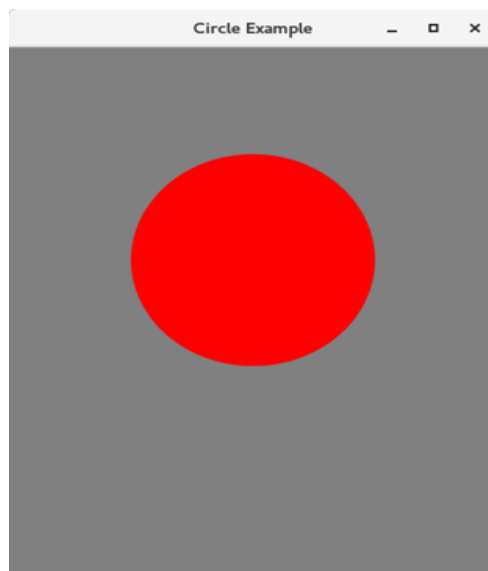
The class properties along with the setter methods and their description are given below in the table.

Property	Description	Setter Methods
centerX	X coordinate of the centre of circle	setCenterX(Double value)
centerY	Y coordinate of the centre of circle	setCenterY(Double value)
radius	Radius of the circle	setRadius(Double value)

Example:

```
package application;  
import javafx.application.Application;  
import javafx.scene.Group;
```

```
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
public class Shape_Example extends Application{
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        primaryStage.setTitle("Circle Example");
        Group group = new Group();
        Circle circle = new Circle();
        circle.setCenterX(200);
        circle.setCenterY(200);
        circle.setRadius(100);
        circle.setFill(Color.RED);
        group.getChildren().addAll(circle);
        Scene scene = new Scene(group,400,500,Color.GRAY);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```



3.13 JavaFX CSS

What is CSS ?

CSS (Cascading Style Sheets) is a design language which is used to enhance the appearance of the web pages without changing its functionality. It only deals with the way, a web page is presented on the web browser. Using CSS, we can define the color, size, font styles, spacing between the paragraph, alignment and many more thing for a web page so that it can look more precise and better. We can also configure the background of the application, layout, design and variety of display for the different size devises.

JavaFX, being the new generation UI library, provides the facility to configure the theme of the application. JavaFX provides the package **javafx.css** which contains all the classes to apply the CSS to the JavaFX application.

Applying CSS to the JavaFX application is similar to applying CSS to the HTML page. In this part of the tutorial, we will discuss styling rules and the steps to invoke them in JavaFX.

Default Style Sheet

JavaFX uses **caspian.css** as the default CSS file. It is found in JavaFX Run time JAR file, **jfxrt.jar**. This style sheet defines the default style rules for the root node and UI controls. This file is located at the path **/jre/lib** under the JDK installation directory. The following command can be used to extract the style sheet from the JAR file.

```
# jar xf jfxrt.jar
# com/sun/javafx/scene/control/skin/caspian/caspian.css
```

Adding Style-sheets to scene

However, JavaFX provides us the facility to override the default style sheet and define our own styles for every node of the application. The Style-sheet we create, must have the extension **.css** and it must be located in the directory where the main class of the application resides.

In JavaFX, there is a specific syntax of applying CSS to the scene. The syntax is given as follows;

```
Scene scene = new Scene(root,500,400);
scene.getStylesheet().add("path/Stylesheet.css");
```

Defining Styles in StyleSheet

A style definition can be given by using the name of the style which is also known as selector and series of the rules that set the properties for the styles. Styling rules are given within the braces. Consider the following example named as **mystyle.css**. It defines the style definition for the each button node used in its container application.

Example

```
.button {
    -fx-font : 14px "serief";
    -fx-padding : 10;
    -fx-background-color : #CCFF99;
```

```
}
```

Selectors

There are various types of styles used in JavaFX. However, each type considers its own conventions regarding selectors. Style class selectors naming conventions are,

1. If the style class selector consists of more than one word, use the hyphen between them.
2. Style class selector names are preceded by a dot(.)

Examples of the selectors are:

1. `.button`
2. `.check-box`
3. `.label`

The style for a particular node can be defined by using the node's ID. This ID can be set using `setId()` method. Use the `#` symbol before the Node_ID to make a style name for that node. For example, the node having id **my_label** can have the following type of selector name.

1. `#my_label`

Defining Rules in Style-sheets

The rules for a style definition assigns values to the properties. There are some conventions for the property names that are given as follows.

1. If the property name consists of more than one word then use hyphen (-) to separate them.
2. Property name for the styles are preceded by `-fx-`.
3. Property name and the value are separated by colon (:).
4. Rules are separated by the semicolon (;).

the example of defining rules for the properties is as follows.

```
-fx-background-color : #333356;
```

```
-fx-font : 16px "serief";
```

There is a special style class named as **.root** defined in javafx. It is applied to the root node of the scene object. Since all the nodes of the application are the children of the root node therefore the style rules applied to this class can be applied to the whole scene graph of the application.

```
.root
```

```
{
```

```
-fx-font-family : "serief";
```

```
-fx-background-color : rgb(225,227,2255);
```

```
}
```

Class Styles

The class styles can be created by adding its definition to our style sheet. For example;

```
.label1{
```

```
-fx-background-color : rgb(123,126,227);
```

```
-fx-padding : 5;
```

```
-fx-text-fill : rgb(245,123,201);
```

```
}
```

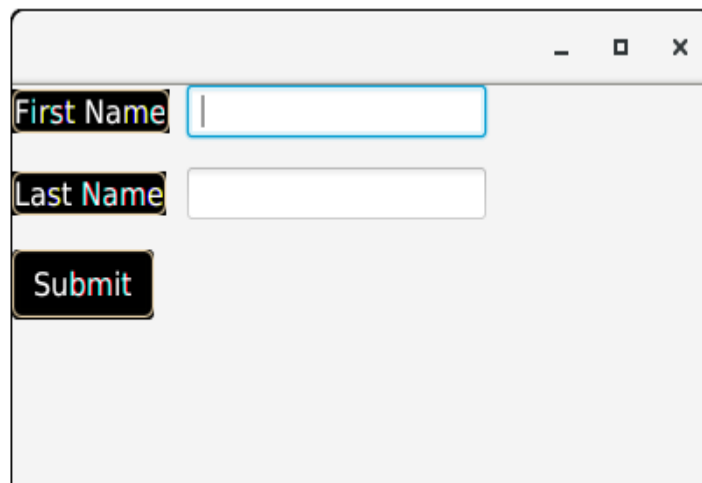
to add the above mentioned style class to the appropriate node, use the **method `getStyleClass().add()`** sequence of methods.

1. `Button button = new Button("SUBMIT");`
2. `button.getStyleClass().add(button1);`

Example:

```
package application;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
public class JavaFX_CSSExample extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        Label first_name=new Label("First Name");
        Label last_name=new Label("Last Name");
        TextField tf1=new TextField();
        TextField tf2=new TextField();
        Button Submit=new Button ("Submit");
        GridPane root=new GridPane();
        root.setHgap(10);
        root.setVgap(15);
        Scene scene = new Scene(root,400,200);
        root.addRow(0, first_name,tf1);
        root.addRow(1, last_name,tf2);
        root.addRow(2, Submit);
        //Adding CSS file to the root
        root.getStylesheets().add("Style.css");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
// style.css
.label
```

```
{
    -fx-background-color:Black;
    -fx-text-fill:white;
    -fx-font-size:16;
    -fx-border-color: Wheat;
    -fx-border-radius: 5;
}
.button
{
    -fx-background-color:Black;
    -fx-font-family:courier_new;
    -fx-text-fill:White;
    -fx-border-color: Wheat;
    -fx-border-radius: 5;
    -fx-font-size:16;
```



3.14 JavaFX Animation

In general, the animation can be defined as the transition which creates the myth of motion for an object. It is the set of transformations applied on an object over the specified duration sequentially so that the object can be shown as it is in motion. This can be done by the rapid display of frames.

In JavaFX, the package **javafx.animation** contains all the classes to apply the animations onto the nodes. All the classes of this package extend the class **javafx.animation.Animation**. JavaFX provides the classes for the transitions like RotateTransition, ScaleTransition, TranslateTransition, FadeTransition, FillTransition, StrokeTransition, etc.

The package **javafx.animation** provides the classes for performing the following transitions.

SN	Transition	Description
1	Rotate Transition	Rotate the Node along one of the axes over the specified duration.
2	Scale Transition	Animate the scaling of the node over the specified duration.
3	Translate Transition	Translate the node from one position to another over the specified duration.
4	Fade Transition	Animate the opacity of the node. It keeps updating the opacity of the node over a specified duration in order to reach a target opacity value
5	Fill Transition	Animate the node's fill color so that the fill color of the node fluctuates between the two color values over the specified duration.
6	Stroke Transition	Animate the node's stroke color so that the stroke color of the node fluctuates between the two color values over the specified duration.
7	Perform the list of transitions on a node in the sequential order.	
8	Parallel Transition	Perform the list of transitions on a node in parallel.
9	Path Transition	Move the node along the specified path over the specified duration.

3.14.1 Steps for applying Animations

1. Create the target node and configure its properties.
Rectangle rect = **new** Rectangle(120,100,100,100);
rect.setFill(Color.RED);
2. Instantiate the respective transition class
RotateTransition rotate = **new** RotateTransition();
3. Set the desired properties like duration, cycle-count, etc. for the transition.
rotate.setDuration(Duration.millis(1000));

```
rotate.setAxis(Rotate.Y_Axis);
rotate.setCycleCount(500);
```

4. Set the target node on which the transition will be applied. Use the following method for this purpose.

```
rotate.setNode(rect);
```
5. Finally, play the transition using the play() method.

```
rotate.play();
```

3.14.2 JavaFX Rotate Transition

This transition is used to apply the rotation transition on the node. It rotates the node along any of the three axes over the specified duration.

RotateTransition is represented by the class **javafx.animation.RotateTransition**. We just need to instantiate this class in order to generate an appropriate RotateTransition.

The properties of the class along with their setter methods are described in the following table.

Property	Description	Setter Methods
Axis	This is a object type property of the class Point3D. This represents the axis of rotate transition.	setAxis(Point3D value)
byAngle	This is a double type property. This represents the angle by which the object will be rotated.	setByAngle(double value)
duration	This is the object type property of the class Duration. This represents the duration of the rotate transition.	setDuration(Duration value)
fromAngle	It is a double type property. It represents the start Angle of the rotate transition.	setFromAngle(double value)
Node	It is an object type property of the class Node. It represents the node on which the rotate transition to be applied.	setNode(Node value)
toAngle	It is a double type property. It represents the stop angle value for the rotate transition.	setToAngle(double value)

There are three constructors in the class.

1. **public RotateTransition()** : creates the new instance of RotateTransition with the default parameters.
2. **public RotateTransition(Duration duration)** : Creates the new instance of RotateTransition with the specified duration value
3. **public RotateTransition(Duration duration, Node node)**: creates the new instance of RotateTransition with the specified duration value and Node on which, it is applied.

Example

In the following example, we have made a rectangle rotating along the Z-axis by 360 degree.

```
package application;
import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.geometry.Point3D;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
import javafx.util.Duration;
public class Rotate_Transition extends Application{
    @Override
    public void start(Stage primaryStage) throws Exception {
        // TODO Auto-generated method stub
        //Creating Rectangle
        Rectangle rect = new Rectangle(200,100,200,200);
        rect.setFill(Color.LIMEGREEN);
        rect.setStroke(Color.HOTPINK);
        rect.setStrokeWidth(5);
        //Instantiating RotateTransition class
        RotateTransition rotate = new RotateTransition();
        //Setting Axis of rotation
        rotate.setAxis(Rotate.Z_AXIS);
        // setting the angle of rotation
        rotate.setByAngle(360);
        //setting cycle count of the rotation
        rotate.setCycleCount(500);
        //Setting duration of the transition
        rotate.setDuration(Duration.millis(1000));
```

```
//the transition will be auto reversed by setting this to true
rotate.setAutoReverse(true);
// transition will be applied
rotate.setNode(rect);
//playing the transition
rotate.play();
//Configuring Group and Scene
Group root = new Group();
root.getChildren().add(rect);
Scene scene = new Scene(root,600,400,Color.BLACK);
primaryStage.setScene(scene);
primaryStage.setTitle("Rotate Transition example");
primaryStage.show();
}
public static void main(String[] args) {
    launch(args);
}
}
```

Chapter 4: Streams and File I/O

4.1 Introduction Streams and File

Data stored in variables and arrays is temporary—it's lost when a local variable goes out of scope or when the program terminates. For long-term retention of data, even after the programs that create the data terminate, computers use files. Java views each file as a **sequential stream of bytes** [1]. The `java.io` package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the `java.io` package supports many data such as primitives, object, localized characters, etc.

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.

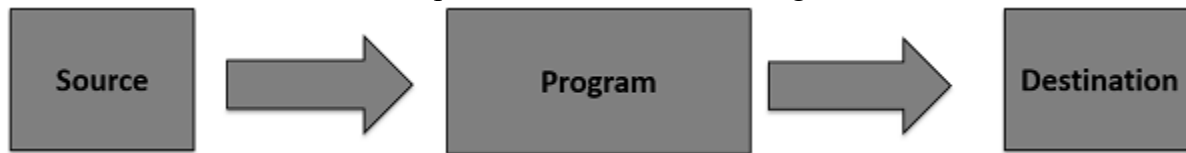


Figure 4. 1 File Stream

Java provides strong but flexible support for I/O related to files and networks. There are different kinds of streams. These are:

Byte Streams

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();
        }
    }
}
```

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();  
    }  
  }  
}
```

4.2 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. 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

```
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();  

```

Enter characters, 'q' to quit.

l
l
e
e
q
q

```
        System.out.print(c);
    } while(c != 'q');
}finally {
    if (cin != null) {
        cin.close();
    }
}
}
```

This program continues to read and output the same character until we press 'q' –

```
$javac ReadConsole.java
```

```
$java ReadConsole
```

4.3 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.

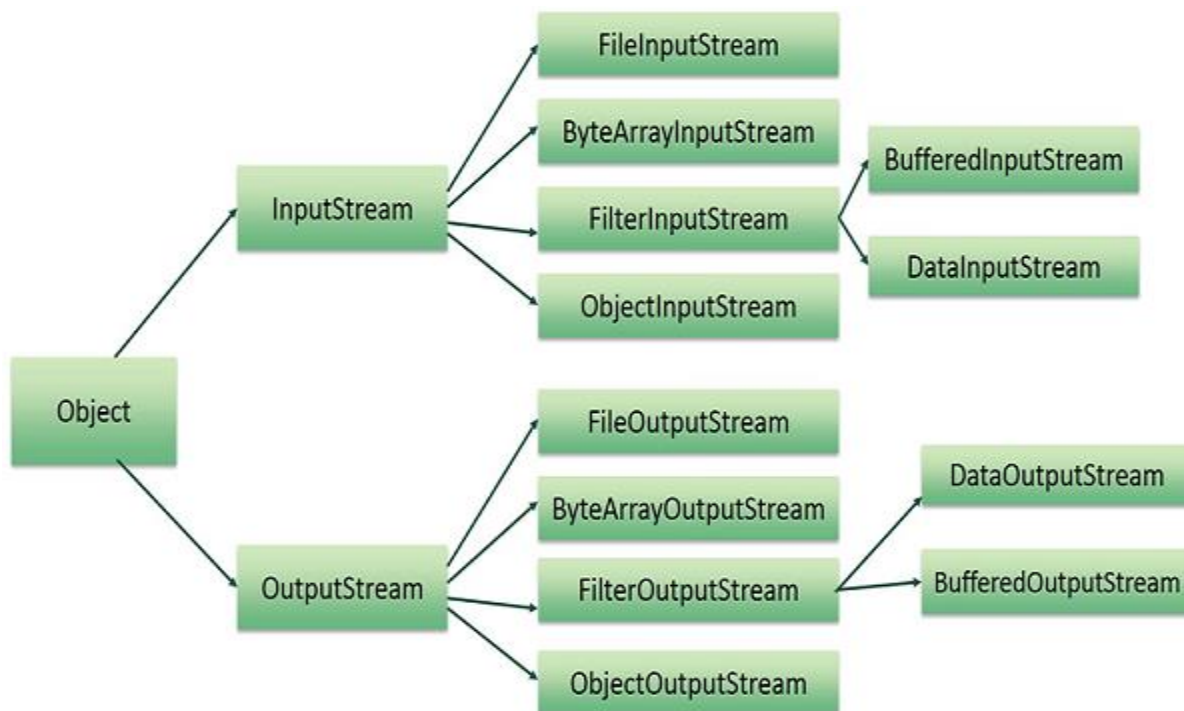


Figure 4. 2 Java Stream classes

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available. Following constructor takes a file name as a string to create an input stream object to read the file

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows –

```
File f = new File("C:/java/hello");
```

```
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Table 4. 1 Methods used in inputStream

Sr.No.	Method & Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{} This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	public int read(byte[] r) throws IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	public int available() throws IOException{} Gives the number of bytes that can be read from this file input stream. Returns an int.

Java BufferedInputStream Class

Java `BufferedInputStream` [class](#) is used to read information from [stream](#). It internally uses buffer mechanism to make the performance fast. The important points about `BufferedInputStream` are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a `BufferedInputStream` is created, an internal buffer [array](#) is created.

Table 4. 2 Java `BufferedInputStream` class methods

Method	Description
<code>int available()</code>	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
<code>int read()</code>	It read the next byte of data from the input stream.
<code>int read(byte[] b, int off, int ln)</code>	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
<code>void close()</code>	It closes the input stream and releases any of the system resources associated with the stream.
<code>void reset()</code>	It repositions the stream at a position the mark method was last called on this input stream.
<code>void mark(int readlimit)</code>	It sees the general contract of the mark method for the input stream.
<code>long skip(long x)</code>	It skips over and discards x bytes of data from the input stream.
<code>boolean markSupported()</code>	It tests for the input stream to support the mark and reset methods.

Example of Java `BufferedInputStream`

Let's see the simple example to read data of [file](#) using `BufferedInputStream`:

```
package com.cs;
import java.io.*;
public class BufferedInputStreamExample{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("D:\\testout.txt");
            BufferedInputStream bin=new BufferedInputStream(fin);
            int i;
            while((i=bin.read())!=-1){
                System.out.print(i+" ");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output. Here are two constructors which can be used to create a **FileOutputStream** object. Following constructor takes a file name as a string to create an input stream object to write the file

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows –

```
File f = new File("C:/java/hello");
```

```
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Table 4. 3 Output Stream Method & Description

Sr.No.	Method & Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.
4	public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.

Example

Following is the example to demonstrate InputStream and OutputStream –

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

    public static void main(String args[]) {

        try {
            byte bWrite [] = {11,21,3,40,5};
            OutputStream os = new FileOutputStream("test.txt");
            for(int x = 0; x < bWrite.length ; x++) {
                os.write( bWrite[x] ); // writes the bytes
            }
            os.close();

            InputStream is = new FileInputStream("test.txt");
            int size = is.available();

            for(int i = 0; i < size; i++) {
                System.out.print((char)is.read() + " ");
            }
            is.close();
        } catch (IOException e) {
            System.out.print("Exception");
        }
    }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

BufferedOutputStream

Java BufferedOutputStream [class](#) is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

OutputStream os= **new** BufferedOutputStream(**new** FileOutputStream("D:\\java\\testout.txt"));

Java BufferedOutputStream class declaration

Let's see the declaration for Java.io.BufferedOutputStream class:

public class BufferedOutputStream **extends** FilterOutputStream

Table 4. 4 Java BufferedOutputStream class methods

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array , starting with the given offset
void flush()	It flushes the buffered output stream.

In this the following example, we are writing the textual information in the BufferedOutputStream object which is connected to the [FileOutputStream object](#). The **flush()** flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```

package com.cs;
import java.io.*;
public class BufferedOutputStreamExample{
    public static void main(String args[])throws Exception{
        FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
        BufferedOutputStream bout=new BufferedOutputStream(fout);
        String s="Welcome to Java.";
        byte b[]=s.getBytes();
        bout.write(b);
        bout.flush();
        bout.close();
        fout.close();
        System.out.println("success");
    }
}

```

4.4 File Management

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- [File Class](#)
- [FileReader Class](#)

- [FileWriter Class](#)

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion, etc.

Getting Path Objects from URIs

An overloaded version of Files static method get uses a URI object to locate the file or directory. A Uniform Resource Identifier (URI) is a more general form of the Uniform Resource Locators (URLs) that are used to locate websites. URIs for locating files vary across operating systems. On Windows platforms, [file://C:/data.txt](#) and the URI identifies the file data.txt stored in the root directory of the C: drive.

The File object represents the actual file/directory on the disk. Following is the list of constructors to create a File object.

Table 4. 5 File Method & Description

Sr.No.	Method & Description
1	File(File parent, String child) This constructor creates a new File instance from a parent abstract pathname and a child pathname string.
2	File(String pathname) This constructor creates a new File instance by converting the given pathname string into an abstract pathname.
3	File(String parent, String child) This constructor creates a new File instance from a parent pathname string and a child pathname string.
4	File(URI uri) This constructor creates a new File instance by converting the given file: URI into an abstract pathname.

Once you have *File* object in hand, then there is a list of helper methods which can be used to manipulate the files.

Table 4. 6 File Helper Methods

Sr.No.	Method & Description
--------	----------------------

1	public String getName() Returns the name of the file or directory denoted by this abstract pathname.
2	public String getParent() Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
3	public File getParentFile() Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
4	public String getPath() Converts this abstract pathname into a pathname string.
5	public boolean isAbsolute() Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise.
6	public String getAbsolutePath() Returns the absolute pathname string of this abstract pathname.
7	public boolean canRead() Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
8	public boolean canWrite() Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
9	public boolean exists()

	Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise.
10	public boolean isDirectory() Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
11	public boolean isFile() Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise.
12	public long lastModified() Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.
13	public long length() Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
14	public boolean createNewFile() throws IOException Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.
15	public boolean delete() Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.

16	public void deleteOnExit() Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
17	public String[] list() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
18	public String[] list(FilenameFilter filter) Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
20	public File[] listFiles() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
21	public File[] listFiles(FileFilter filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
22	public boolean mkdir() Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise.
23	public boolean mkdirs() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
24	public boolean renameTo(File dest) Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise.

25	public boolean setLastModified(long time) Sets the last-modified time of the file or directory named by this abstract pathname. Returns true if and only if the operation succeeded; false otherwise.
26	public boolean setReadOnly() Marks the file or directory named by this abstract pathname so that only read operations are allowed. Returns true if and only if the operation succeeded; false otherwise.
27	public static File createTempFile(String prefix, String suffix, File directory) throws IOException Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. Returns an abstract pathname denoting a newly-created empty file.
28	public static File createTempFile(String prefix, String suffix) throws IOException Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. Invoking this method is equivalent to invoking createTempFile(prefix, suffix, null). Returns abstract pathname denoting a newly-created empty file.
29	public int compareTo(File pathname) Compares two abstract pathnames lexicographically. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
30	public int compareTo(Object o) Compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
31	public boolean equals(Object obj) Tests this abstract pathname for equality with the given object. Returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.

32

public String toString()

Returns the pathname string of this abstract pathname. This is just the string returned by the `getPath()` method.

Example

Following is an example to demonstrate File object –

```
package com.tutorialspoint;
import java.io.File;

public class FileDemo {

    public static void main(String[] args) {
        File f = null;
        String[] strs = {"test1.txt", "test2.txt"};
        try { // for each string in string array
            for(String s:strs ) {
                // create new file
                f = new File(s);
                // true if the file is executable
                boolean bool = f.canExecute();
                // find the absolute path
                String a = f.getAbsolutePath();
                // prints absolute path
                System.out.print(a);
                // prints
                System.out.println(" is executable: "+ bool);
            }
        } catch (Exception e) {
            // if any I/O error occurs
            e.printStackTrace();
        }
    }
}
```

File Reader Class

This class inherits from the `InputStreamReader` class. `FileReader` is used for reading streams of characters. This class has several constructors to create required objects. Following is the list of constructors provided by the `FileReader` class. Once you have `FileReader` object in hand then there is a list of helper methods which can be used to manipulate the files.

Table 4. 7 File Reader Class Methos

Sr.No.	Method & Description
1	<code>public int read() throws IOException</code> Reads a single character. Returns an int, which represents the character read.
2	<code>public int read(char [] c, int offset, int len)</code> Reads characters into an array. Returns the number of characters read.

Following is an example to demonstrate class

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

    public static void main(String args[])throws IOException {
        File file = new File("Hello1.txt");

        // creates the file
        file.createNewFile();

        // creates a FileWriter Object
        FileWriter writer = new FileWriter(file);

        // Writes the content to the file
        writer.write("This\n is\n an\n example\n");
        writer.flush();
        writer.close();

        // Creates a FileReader Object
        FileReader 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
        fr.close();
    }
}
```

Output

This
is
an
example

```
}
```

Directories in Java

A directory is a **File** which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on **File** object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories –

- The **mkdir()** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the **File** object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory –

Example

```
import java.io.File;
public class CreateDir {

    public static void main(String args[]) {
        String dirname = "/tmp/user/java/bin";
        File d = new File(dirname);

        // Create directory now.
        d.mkdirs();
    }
}
```

Note – Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list()** method provided by **File** object to list down all the files and directories available in a directory as follows –

Example

```
import java.io.File;
public class ReadDir {

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

```
File file = null;
String[] paths;

try {
    // create new file object
    file = new File("/tmp");

    // array of files and directory
    paths = file.list();

    // for each name in the path array
    for(String path:paths) {
        // prints filename and directory name
        System.out.println(path);
    }
} catch (Exception e) {
    // if any error occurs
    e.printStackTrace();
}
}
```

Output

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

Chapter 5: Multi-Threading in Java

5.1 Introduction to Multi-threading

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. **Multithreading in Java** is a process of executing multiple threads simultaneously. However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation, etc.

A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs. Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Advantages of Java Multithreading

- It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- You **can perform many operations together, so it saves time**.
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another requires some time for saving and loading [registers](#), memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

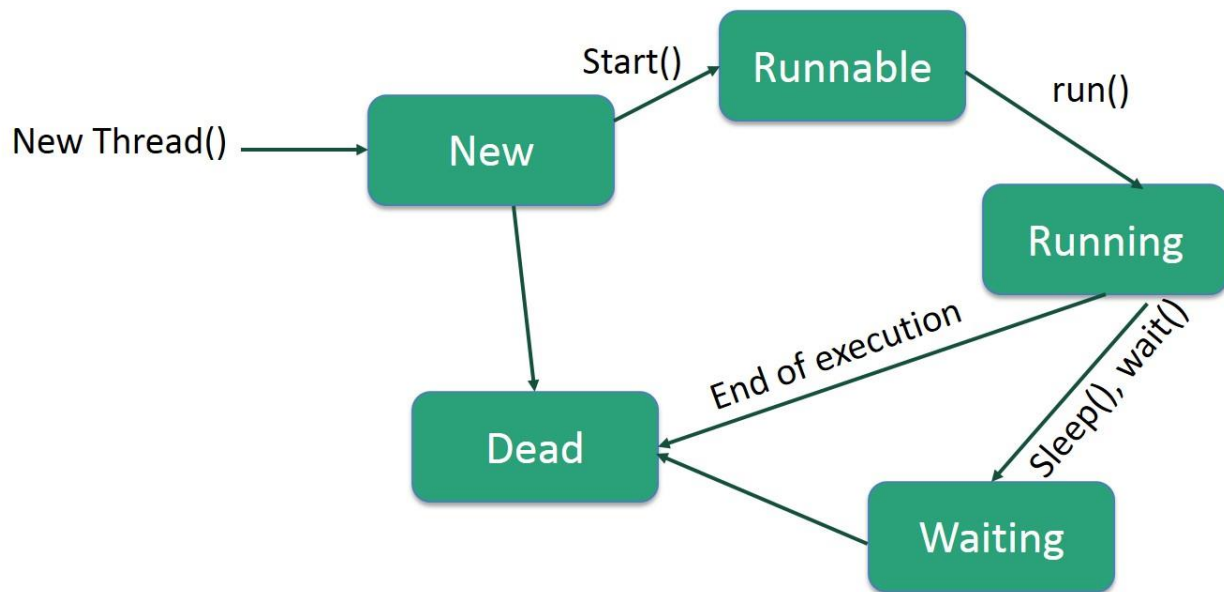
5.2 Difference between Process and Thread

S.NO	Process	Thread
1.	Process means any program is in execution.	Thread means a segment of a process.
2.	The process takes more time to terminate.	The thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
6.	Multiprogramming holds the concepts of multi-process.	We don't need multi programs in action for multiple threads because a single process consists of multiple threads.
7.	The process is isolated.	Threads share memory.
8.	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
9.	Process switching uses an interface in an operating system.	Thread switching does not require calling an operating system and causes an interrupt to the kernel.
10.	If one process is blocked then it will not affect the execution of other processes	If a user-level thread is blocked, then all other user-level threads are blocked.

S.NO	Process	Thread
11.	The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.
12.	Changes to the parent process do not affect child processes.	Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behavior of the other threads of the process.
14.	The process does not share data with each other.	Threads share data with each other.

5.3 Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle:

- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

5.4 The Implementation Threads in java

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

5.4.1 Create a Thread by Extending a Thread Class

The first way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1: You will need to override run() method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method.

public void run()

Step 2: Once Thread object is created, you can start it by calling start() method, which executes a call to run() method. Following is a simple syntax of start() method.

void start();

Example:

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
} }
```

5.4.2 Create a Thread by Implementing a Runnable Interface

If your class is intended to be executed as a thread then you can achieve this by implementing a **Runnable** interface. You will need to follow three basic steps –

Step 1: As a first step, you need to implement a run() method provided by a **Runnable** interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method:

public void run()

Step 2: As a second step, you will instantiate a **Thread** object using the following constructor – **Thread(Runnable threadObj, String threadName);**

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3: Once a Thread object is created, you can start it by calling **start()** method, which executes a call to **run()** method.

void start();

Runnable Example

// Java code for thread creation by implementing

// the Runnable Interface

```
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Output

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```

5.5 The Mechanisms for handling Multiple Threads

5.5.1 Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The `java.lang.Thread.getPriority()` method returns the priority of the given thread.

public final void setPriority(int newPriority): The `java.lang.Thread.setPriority()` method updates or assign the priority of the thread to `newPriority`. The method throws `IllegalArgumentException` if the value `newPriority` goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

Default priority of a thread is 5 (`NORM_PRIORITY`). The value of `MIN_PRIORITY` is 1 and the value of `MAX_PRIORITY` is 10.

// Importing the required classes

import java.lang.*;

public class ThreadPriorityExample **extends** Thread

{

// Method 1

// Whenever the start() method is called by a thread // the run() method is invoked

public void run()

{

// the print statement

System.out.println("Inside the run() method");

}

// the main method

public static void main(String args[])

{

// Creating threads with the help of ThreadPriorityExample class

ThreadPriorityExample th1 = **new** ThreadPriorityExample();

ThreadPriorityExample th2 = **new** ThreadPriorityExample();

```
ThreadPriorityExample th3 = new ThreadPriorityExample();
// using the getPriority() method
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
// passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());
// Main thread
// Displaying name of the currently executing thread
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName(
));
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// Priority of the main thread is 10 now
Thread.currentThread().setPriority(10);
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}
}
```

5.5.2 Java - Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement –

Syntax

```
synchronized(objectidentifier) {
    // Access shared variables and other shared resources
}
```

Here, the **object identifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

Multithreading Example without Synchronization

Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces a different result based on CPU availability to a thread.

```
class PrintDemo {
    public void printCount() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Counter --- " + i );
            }
        } catch (Exception e) {
            System.out.println("Thread interrupted.");
        }
    }
}

class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;
    PrintDemo PD;
    ThreadDemo( String name, PrintDemo pd) {
        threadName = name;
        PD = pd;
    }
    public void run() {
        PD.printCount();
        System.out.println("Thread " + threadName + " exiting.");
    }
    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}
```

```
public class TestThread {
    public static void main(String args[]) {
        PrintDemo PD = new PrintDemo();
        ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
        ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
        T1.start();
        T2.start();
        // wait for threads to end
        try {
            T1.join();
            T2.join();
        } catch ( Exception e) {
            System.out.println("Interrupted");
        }
    }
}
```

The output of the above code

```
Starting Thread - 1
Starting Thread - 2
Counter --- 5
Counter --- 4
Counter --- 3
Counter --- 5
Counter --- 2
Counter --- 1
Counter --- 4
Thread Thread - 1 exiting.
Counter --- 3
Counter --- 2
Counter --- 1
Thread Thread - 2 exiting.
```

Multithreading Example with Synchronization

Here is the same example which prints counter value in sequence and every time we run it, it produces the same result.

```
class PrintDemo {
    public void printCount() {
        try {
```

```
        for(int i = 5; i > 0; i--) {
            System.out.println("Counter --- " + i );
        }
    } catch (Exception e) {
        System.out.println("Thread interrupted.");
    }
}
}

class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;
    PrintDemo PD;
    ThreadDemo( String name, PrintDemo pd) {
        threadName = name;
        PD = pd;
    }
    public void run() {
        synchronized(PD) {
            PD.printCount();
        }
        System.out.println("Thread " + threadName + " exiting.");
    }
    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {
    public static void main(String args[]) {
        PrintDemo PD = new PrintDemo();
        ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
        ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
        T1.start();
        T2.start();
        // wait for threads to end
        try {
            T1.join();
        }
    }
}
```



```
        T2.join();
    } catch ( Exception e) {
        System.out.println("Interrupted");
    }
}
}
```

The output of the above code

```
Starting Thread - 1
Starting Thread - 2
Counter --- 5
Counter --- 4
Counter --- 3
Counter --- 2
Counter --- 1
Thread Thread - 1 exiting.
Counter --- 5
Counter --- 4
Counter --- 3
Counter --- 2
Counter --- 1
Thread Thread - 2 exiting.
```

Chapter 6: Networking in Java

6.1 Introduction to network programming

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The `java.net` package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The `java.net` package provides support for the two common network protocols –

- **TCP** – TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP** – UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects –

- **Socket Programming** – This is the most widely used concept in Networking and it has been explained in very detail.
- **URL Processing** – This would be covered separately.

6.2 Socket Programming

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server. When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

The `java.net.Socket` class represents a socket, and the `java.net.ServerSocket` class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets –

- The server instantiates a `ServerSocket` object, denoting which port number communication is to occur on.
- The server invokes the `accept()` method of the `ServerSocket` class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a `Socket` object, specifying the server name and the port number to connect to.
- The constructor of the `Socket` class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a `Socket` object capable of communicating with the server.

- On the server side, the `accept()` method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an `OutputStream` and an `InputStream`. The client's `OutputStream` is connected to the server's `InputStream`, and the client's `InputStream` is connected to the server's `OutputStream`.

TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets

6.2.1 Implementing Socket Programming

We can implement socket programming using two `java.net` package classes. These are **`ServerSocket`** Class and **`Socket`** classes and their Methods.

The **`java.net.ServerSocket`** class is used by server applications to obtain a port and listen for client requests.

The `ServerSocket` class has four constructors –

Sr.No.	Method & Description
1	<code>public ServerSocket(int port) throws IOException</code> Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	<code>public ServerSocket(int port, int backlog) throws IOException</code> Similar to the previous constructor, the <code>backlog</code> parameter specifies how many incoming clients to store in a wait queue.
3	<code>public ServerSocket(int port, int backlog, InetAddress address) throws IOException</code> Similar to the previous constructor, the <code>InetAddress</code> parameter specifies the local IP address to bind to. The <code>InetAddress</code> is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on.
4	<code>public ServerSocket() throws IOException</code> Creates an unbound server socket. When using this constructor, use the <code>bind()</code> method when you are ready to bind the server socket.

If the `ServerSocket` constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Following are some of the common methods of the ServerSocket class –

Sr.No.	Method & Description
1	public int getLocalPort() Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2	public Socket accept() throws IOException Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely.
3	public void setSoTimeout(int timeout) Sets the time-out value for how long the server socket waits for a client during the accept().
4	public void bind(SocketAddress host, int backlog) Binds the socket to the specified server and port in the SocketAddress object. Use this method if you have instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and the server, and communication can begin.

Socket Class Methods

The **java.net.Socket** class represents the socket that both the client and the server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server –

Sr.No.	Method & Description
1	public Socket(String host, int port) throws UnknownHostException, IOException. This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.

2	public Socket(InetAddress host, int port) throws IOException This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.
3	public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException. Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException. This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String.
5	public Socket() Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and the server have a Socket object, so these methods can be invoked by both the client and the server.

Sr.No.	Method & Description
1	public void connect(SocketAddress host, int timeout) throws IOException This method connects the socket to the specified host. This method is needed only when you instantiate the Socket using the no-argument constructor.
2	public InetAddress getInetAddress() This method returns the address of the other computer that this socket is connected to.
3	public int getPort() Returns the port the socket is bound to on the remote machine.

4	public int getLocalPort() Returns the port the socket is bound to on the local machine.
5	public SocketAddress getRemoteSocketAddress() Returns the address of the remote socket.
6	public InputStream getInputStream() throws IOException Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7	public OutputStream getOutputStream() throws IOException Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket.
8	public void close() throws IOException Closes the socket, which makes this Socket object no longer capable of connecting again to any server.

InetAddress Class Methods

This class represents an Internet Protocol (IP) address. Here are following usefull methods which you would need while doing socket programming –

Sr.No.	Method & Description
1	static InetAddress getByAddress(byte[] addr) Returns an InetAddress object given the raw IP address.
2	static InetAddress getByAddress(String host, byte[] addr) Creates an InetAddress based on the provided host name and IP address.
3	static InetAddress getByName(String host) Determines the IP address of a host, given the host's name.
4	String getHostAddress()

	Returns the IP address string in textual presentation.
5	String getHostName() Gets the host name for this IP address.
6	static InetAddress InetAddress getLocalHost() Returns the local host.
7	String toString() Converts this IP address to a String.

Socket Client Example

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

Example

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;
public class GreetingClient {
    public static void main(String [] args) {
        String serverName = args[0];
        int port = Integer.parseInt(args[1]);
        try {
            System.out.println("Connecting to " + serverName + " on port " + port);
            Socket client = new Socket(serverName, port);
            System.out.println("Just connected to " + client.getRemoteSocketAddress());
            OutputStream outToServer = client.getOutputStream();
            DataOutputStream out = new DataOutputStream(outToServer);
            out.writeUTF("Hello from " + client.getLocalSocketAddress());
            InputStream inFromServer = client.getInputStream();
            DataInputStream in = new DataInputStream(inFromServer);
            System.out.println("Server says " + in.readUTF());
            client.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
}
```

Socket Server Example

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument –

Example

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;
public class GreetingServer extends Thread {
    private ServerSocket serverSocket;
    public GreetingServer(int port) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(10000);
    }
    public void run() {
        while(true) {
            try {
                System.out.println("Waiting for client on port " +
                    serverSocket.getLocalPort() + "...");
                Socket server = serverSocket.accept();
                System.out.println("Just connected to " + server.getRemoteSocketAddress());
                DataInputStream in = new DataInputStream(server.getInputStream());
                System.out.println(in.readUTF());
                DataOutputStream out = new DataOutputStream(server.getOutputStream());
                out.writeUTF("Thank you for connecting to " + server.getLocalSocketAddress()
                    + "\nGoodbye!");
                server.close();
            } catch (SocketTimeoutException s) {
                System.out.println("Socket timed out!");
                break;
            } catch (IOException e) {
                e.printStackTrace();
                break;
            }
        }
    }
    public static void main(String [] args) {
        int port = Integer.parseInt(args[0]);
        try {
```



```
    Thread t = new GreetingServer(port);  
    t.start();  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}  
}
```

Compile the client and the server and then start the server as follows –

```
$ java GreetingServer 6066
```

Waiting for client on port 6066...

Check the client program as follows –

Output

```
$ java GreetingClient localhost 6066  
Connecting to localhost on port 6066  
Just connected to localhost/127.0.0.1:6066  
Server says Thank you for connecting to  
/127.0.0.1:6066  
Goodbye!
```

6.3 Remote Method Invocation

RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

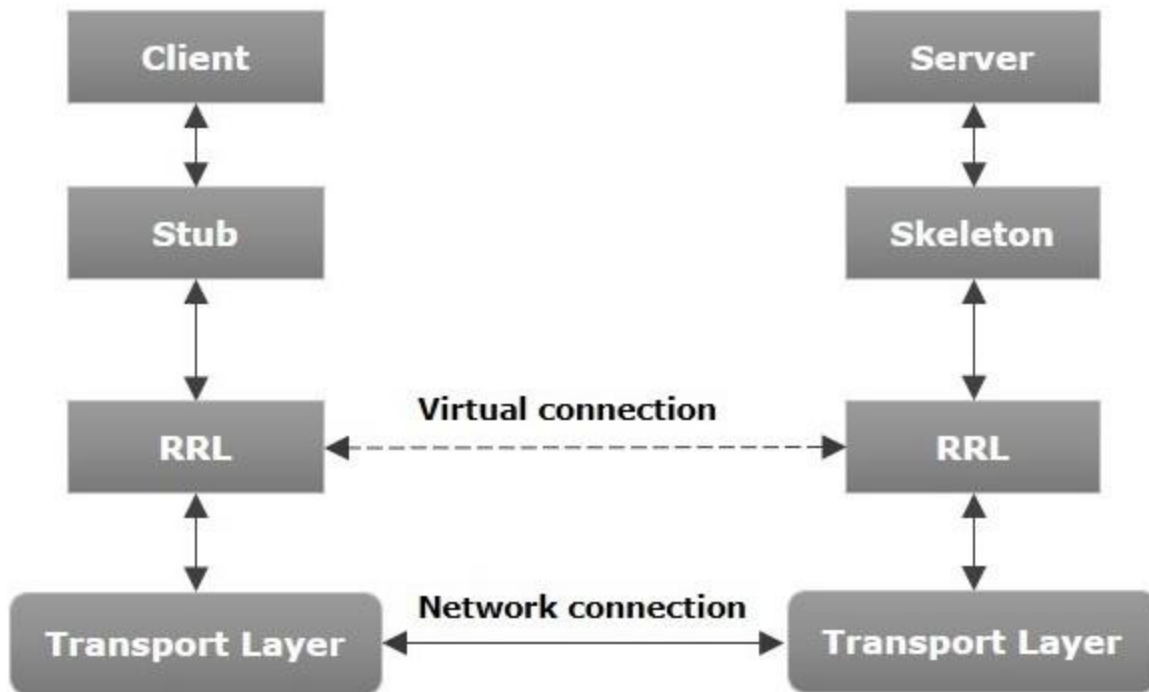
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

6.3.1 Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

- **Transport Layer** – This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** – This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** – It is the layer which manages the references made by the client to the remote object.

6.3.2 Implementing RMI Application

The following points summarize how an RMI application implemented –

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

6.3.3 Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of

primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

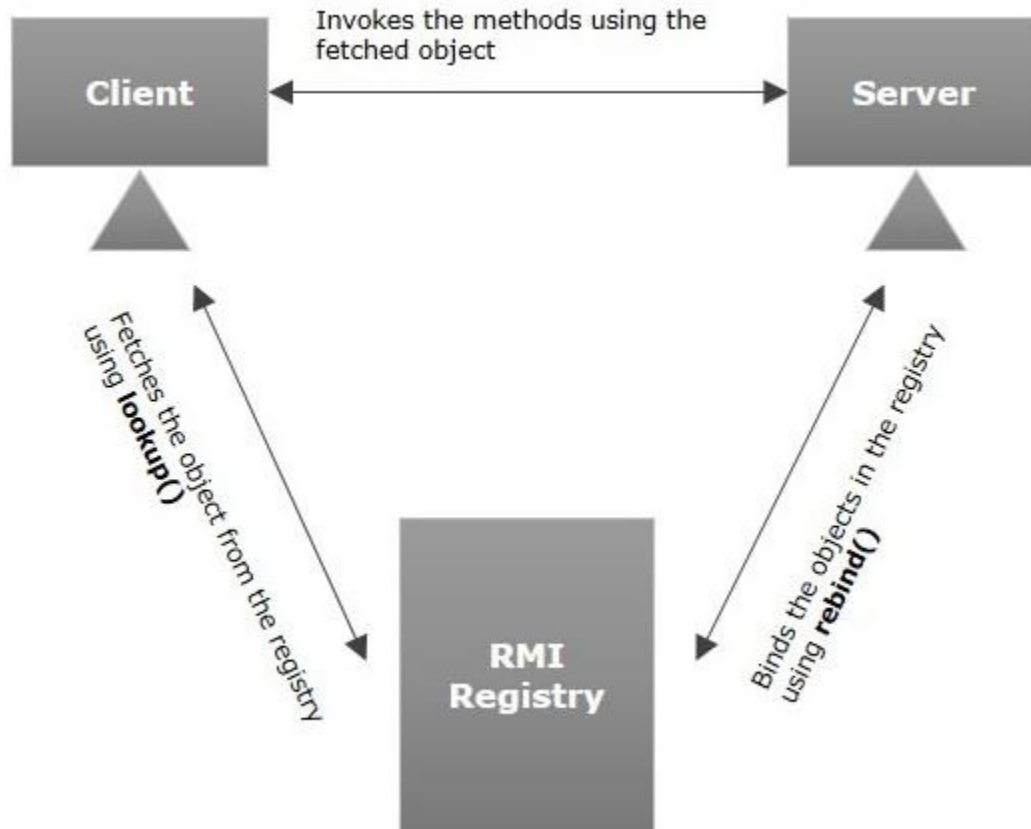
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

6.4 RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** or **reBind()** methods). These are registered using a unique name known as **bind name**.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).

The following illustration explains the entire process –



To write an RMI Java application, you would have to follow the steps given below –

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program

- Compile the application
- Execute the application

6.4.1 Defining the Remote Interface

A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface.

To create a remote interface –

- Create an interface that extends the predefined interface **Remote** which belongs to the package.
- Declare all the business methods that can be invoked by the client in this interface.
- Since there is a chance of network issues during remote calls, an exception named **RemoteException** may occur; throw it.

Following is an example of a remote interface. Here we have defined an interface with the name **Hello** and it has a method called **printMsg()**.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// Creating Remote interface for our application
public interface Hello extends Remote {
    void printMsg() throws RemoteException;
}
```

6.4.2 Developing the Implementation Class (Remote Object)

We need to implement the remote interface created in the earlier step. (We can write an implementation class separately or we can directly make the server program implement this interface.)

To develop an implementation class –

- Implement the interface created in the previous step.
- Provide implementation to all the abstract methods of the remote interface.

Following is an implementation class. Here, we have created a class named **ImplExample** and implemented the interface **Hello** created in the previous step and provided **body** for this method which prints a message.

```
// Implementing the remote interface
public class ImplExample implements Hello {

    // Implementing the interface method
    public void printMsg() {
        System.out.println("This is an example RMI program");
    }
}
```

```
}
```

6.4.3 Developing the Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMIregistry**.

To develop a server program –

- Create a client class from where you want invoke the remote object.
- **Create a remote object** by instantiating the implementation class as shown below.
- Export the remote object using the method **exportObject()** of the class named **UnicastRemoteObject** which belongs to the package **java.rmi.server**.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Bind the remote object created to the registry using the **bind()** method of the class named **Registry**. To this method, pass a string representing the bind name and the object exported, as parameters.

Following is an example of an RMI server program.

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Server extends ImplExample {
    public Server() {}
    public static void main(String args[]) {
        try {
            // Instantiating the implementation class
            ImplExample obj = new ImplExample();
            // Exporting the object of implementation class
            // (here we are exporting the remote object to the stub)
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            // Binding the remote object (stub) in the registry
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

6.4.4 Developing the Client Program

Write a client program in it, fetch the remote object and invoke the required method using this object.

To develop a client program –

- Create a client class from where your intended to invoke the remote object.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Fetch the object from the registry using the method **lookup()** of the class **Registry** which belongs to the package **java.rmi.registry**.
To this method, you need to pass a string value representing the bind name as a parameter. This will return you the remote object.
- The **lookup()** returns an object of type remote, down cast it to the type Hello.
- Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
    private Client() {}
    public static void main(String[] args) {
        try {
            // Getting the registry
            Registry registry = LocateRegistry.getRegistry(null);
            // Looking up the registry for the remote object
            Hello stub = (Hello) registry.lookup("Hello");
            // Calling the remote method using the obtained object
            stub.printMsg();
            // System.out.println("Remote method invoked");
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

6.4.5 Compiling the Application

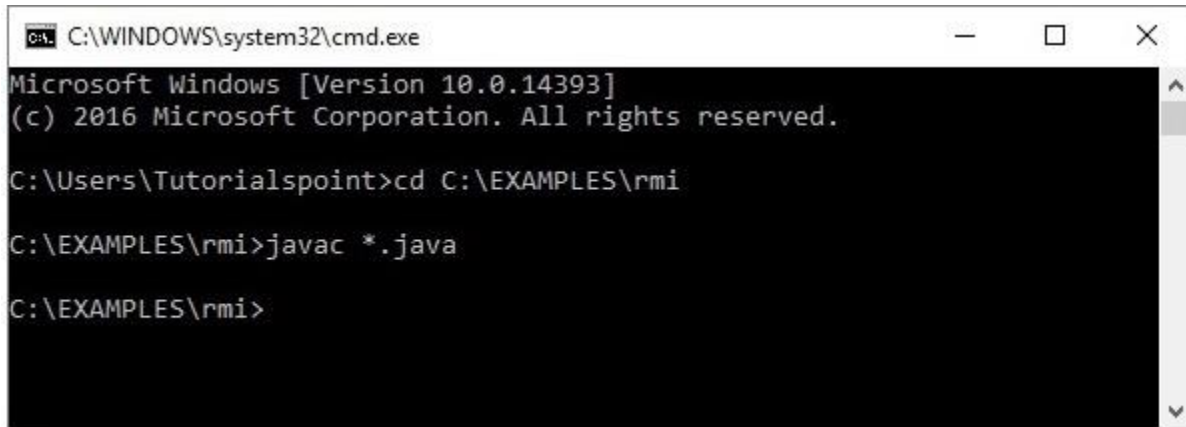
To compile the application –

- Compile the Remote interface.
- Compile the implementation class.

- Compile the server program.
- Compile the client program.

Open the folder where you have stored all the programs and compile all the Java files as shown below.

Javac *.java



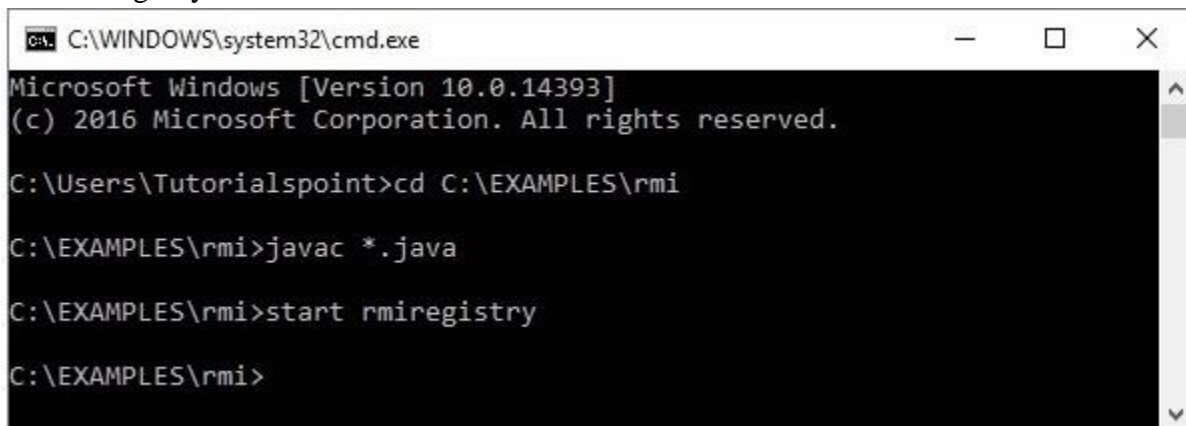
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>javac *.java
C:\EXAMPLES\rmi>
```

6.4.6 Executing the Application

Step 1 – Start the **rmi** registry using the following command.

start rmiregistry



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>javac *.java
C:\EXAMPLES\rmi>start rmiregistry
C:\EXAMPLES\rmi>
```

This will start an **rmi** registry on a separate window as shown below.



Step 2 – Run the server class file as shown below.

Java Server

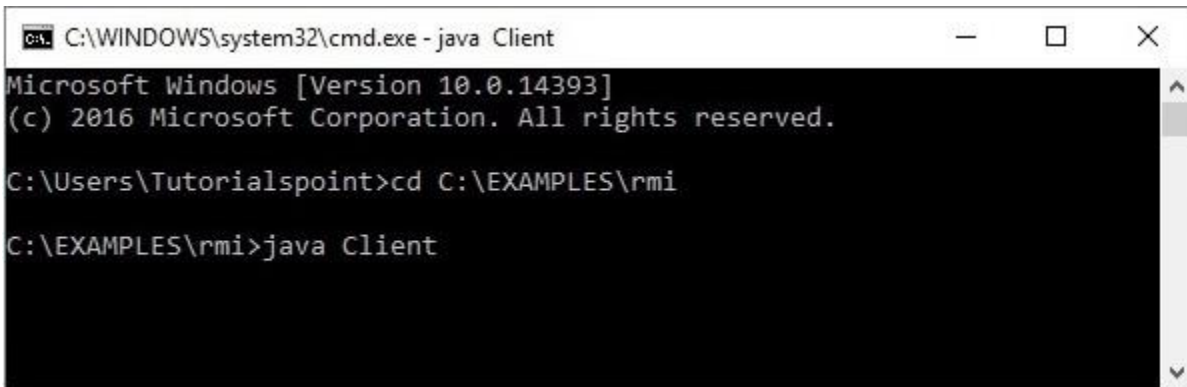


```
C:\WINDOWS\system32\cmd.exe - java Server

C:\EXAMPLES\rmi>java Server
Server ready
```

Step 3 – Run the client class file as shown below.

java Client

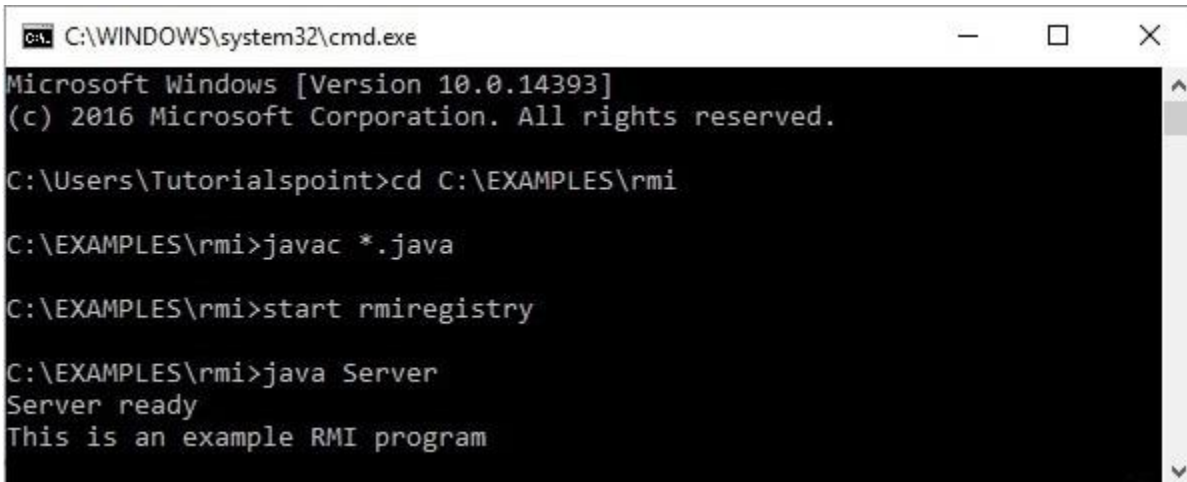


```
C:\WINDOWS\system32\cmd.exe - java Client

Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>java Client
```

Verification – As soon you start the client, you would see the following output in the server.



```
C:\WINDOWS\system32\cmd.exe

Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Tutorialspoint>cd C:\EXAMPLES\rmi
C:\EXAMPLES\rmi>javac *.java
C:\EXAMPLES\rmi>start rmiregistry
C:\EXAMPLES\rmi>java Server
Server ready
This is an example RMI program
```


Chapter 7: Java Database Connectivity

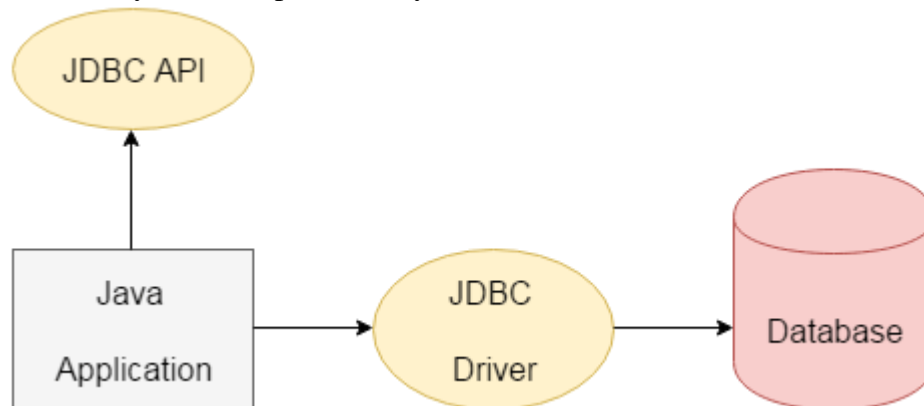
7.1 Introduction to JDBC

A database is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A database management system (DBMS) provides mechanisms for storing, organizing, retrieving and modifying data for many users. Database management systems allow for the access and storage of data without concern for the internal representation of data. A relational database is a logical representation of data that allows the data to be accessed without consideration of its physical structure. A relational database stores data in tables.

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc. We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface

- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

7.2 Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language). We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

7.3 Structured Query Language (SQL)

Structured Query Language (SQL) is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries. SQL is supported by almost any database you will likely use, and it allows you to write database code independently of the underlying database.

Create Database

The CREATE DATABASE statement is used for creating a new database. The syntax is –
SQL> CREATE DATABASE DATABASE_NAME;

Example

The following SQL statement creates a Database named EMP –

```
SQL> CREATE DATABASE EMP;
```

Drop Database

The DROP DATABASE statement is used for deleting an existing database. The syntax is –
SQL> DROP DATABASE DATABASE_NAME;

Note – To create or drop a database you should have administrator privilege on your database server. Be careful, deleting a database would loss all the data stored in the database.

Create Table

The CREATE TABLE statement is used for creating a new table. The syntax is –

```
SQL> CREATE TABLE table_name
(
    column_name column_data_type,
    column_name column_data_type,
    column_name column_data_type
    ...
);
```

Example

The following SQL statement creates a table named Employees with four columns –

```
SQL> CREATE TABLE Employees
(
    id INT NOT NULL,
    age INT NOT NULL,
    first VARCHAR(255),
    last VARCHAR(255),
    PRIMARY KEY ( id )
);
```

Drop Table

The DROP TABLE statement is used for deleting an existing table. The syntax is –

```
SQL> DROP TABLE table_name;
```

Example

The following SQL statement deletes a table named Employees –

```
SQL> DROP TABLE Employees;
```

INSERT Data

The syntax for INSERT, looks similar to the following, where column1, column2, and so on represents the new data to appear in the respective columns –

```
SQL> INSERT INTO table_name VALUES (column1, column2, ...);
```

Example

The following SQL INSERT statement inserts a new row in the Employees database created earlier –

```
SQL> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
```

SELECT Data

The SELECT statement is used to retrieve data from a database. The syntax for SELECT is –

```
SQL> SELECT column_name, column_name, ...  
      FROM table_name  
      WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL statement selects the age, first and last columns from the Employees table, where id column is 100 –

```
SQL> SELECT first, last, age  
      FROM Employees  
      WHERE id = 100;
```

The following SQL statement selects the age, first and last columns from the Employees table where *first* column contains *Zara* –

```
SQL> SELECT first, last, age  
      FROM Employees  
      WHERE first LIKE '%Zara%';
```

UPDATE Data

The UPDATE statement is used to update data. The syntax for UPDATE is –

```
SQL> UPDATE table_name  
      SET column_name = value, column_name = value, ...  
      WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL UPDATE statement changes the age column of the employee whose id is 100 –

```
SQL> UPDATE Employees SET age=20 WHERE id=100;
```

DELETE Data

The DELETE statement is used to delete data from tables. The syntax for DELETE is –

```
SQL> DELETE FROM table_name WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=, and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL DELETE statement deletes the record of the employee whose id is 100 –

```
SQL> DELETE FROM Employees WHERE id=100;
```

7.4 Steps of Java Database Connectivity

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

Java Database Connectivity



7.4.1 JDBC Driver

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server. For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The *Java.sql* package that ships with JDK, contains various classes with their behaviours defined and their actual implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

JDBC Drivers Types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below –

Type 1 – JDBC-ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database. When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available. The **JDBC-ODBC Bridge** that comes with **JDK 1.2** is a good example of this kind of driver.

Type 2 – JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine. If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3 – JDBC-Net pure Java

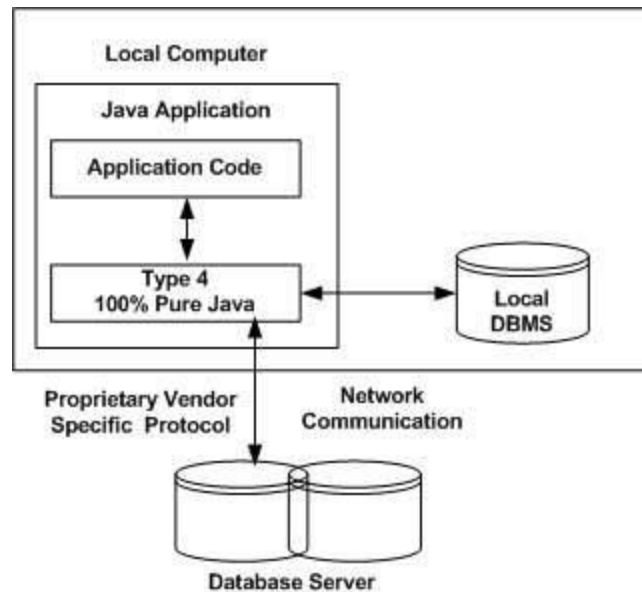
In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server. This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

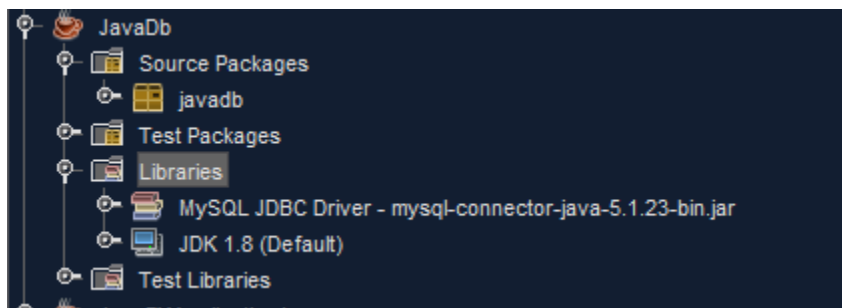
Type 4 – 100% Pure Java

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.



Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4. If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver. Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database. The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

7.4.2 Establishing Connection

After you've installed the appropriate driver, it is time to establish a database connection using JDBC. The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps –

- **Import JDBC Packages** – Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver** – This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.

- **Database URL Formulation** – This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object** – Finally, code a call to the *DriverManager* object's *getConnection()* method to establish actual database connection.

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code –

```
import java.sql.* ; // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces. You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - **Class.forName()**

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it.

The following example uses **Class.forName()** to register the Oracle driver –

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}
```

You can use **newInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows –

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}
catch(IllegalAccessException ex) {
```



```

    System.out.println("Error: access problem while loading!");
    System.exit(2);
catch(InstantiationException ex) {
    System.out.println("Error: unable to instantiate driver!");
    System.exit(3);
}

```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method. You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver –

```

try {
    Driver myDriver = new oracle.jdbc.driver.OracleDriver();
    DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
    System.out.println("Error: unable to load driver class!");
    System.exit(1);
}

```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods –

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database. Formulating a database URL is where most of the problems associated with establishing a connection occurs. Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql:// hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@ hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2: hostname:port Number/databaseName

Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds: hostname: port Number/databaseName
--------	---------------------------	---------------------------------------------------------------

All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object. The most commonly used form of **getConnection()** requires you to pass a database URL, a *username*, and a *password* –

Now you have to call **getConnection()** method with appropriate username and password to get a **Connection** object as follows –

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not. To close the above opened connection, you should call **close()** method as follows –

conn.close();

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

7.4.3 DBC - Statements, PreparedStatement and CallableStatement

Once a connection is obtained we can interact with the database. The *JDBC Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.

PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's **createStatement()** method, as in the following example –

```
Statement stmt = null;
try {
    stmt = conn.createStatement();
    ...
}
catch (SQLException e) {
    ...
}
finally {
    stmt.close();
}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL) –** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL) –** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
```

```
String SQL = "Update Employees SET age = ? WHERE id = ?";
pstmt = conn.prepareStatement(SQL);
...
}
catch (SQLException e) {
    ...
}
finally {
    pstmt.close();
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an **SQLException**.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) **execute()**, (b) **executeQuery()**, and (c) **executeUpdate()** also work with the **PreparedStatement** object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a **Statement** object **.close ()** for the same reason you should also close the **PreparedStatement** object.

7.4.4 ResultSet

The SQL statements that read data from a database query, return the data in a result set. The **SELECT** statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A **ResultSet** object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a **ResultSet** object.

The methods of the **ResultSet** interface can be broken down into three categories –

- **Navigational methods** – Used to move the cursor around.
- **Get methods** – Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods** – Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet –

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable. The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions –

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –

S.N.	Methods & Description
1	public int getInt(String columnName) throws SQLException Returns the int in the current row in the column named columnName.
2	public int getInt(int columnIndex) throws SQLException Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

The ResultSet interface contains a collection of **update methods** for updating the data of a result set. As with the get methods, there are two update methods for each data type –

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods –

S.N.	Methods & Description
1	public void updateString(int columnIndex, String s) throws SQLException Changes the String in the specified column to the value of s.
2	public void updateString(String columnName, String s) throws SQLException

	Similar to the previous method, except that the column is specified by its name instead of its index.
--	-------------------------------------------------------------------------------------------------------

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	public void updateRow() Updates the current row by updating the corresponding row in the database.
2	public void deleteRow() Deletes the current row from the database
3	public void refreshRow() Refreshes the data in the result set to reflect any recent changes in the database.
4	public void cancelRowUpdates() Cancels any updates made on the current row.
5	public void insertRow() Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

7.5 JDBC Sample Code

```
mysql> use wsu;
mysql> drop table Employees;
Query OK, 0 rows affected (0.08 sec)
mysql> create table Employees
-> (
-> id int primary key auto_increment,
-> age int not null,
-> first varchar (255),
-> last varchar (255)
-> );
```

```
Query OK, 0 rows affected (0.08 sec)
mysql>
```

Finally you create few records in Employee table as follows –

```
mysql> INSERT INTO Employees(AGE, FIRST, LAST) VALUES (18, Dawit, Uta);
Query OK, 1 row affected (0.05 sec)
```

```
mysql> INSERT INTO Employees(AGE, FIRST, LAST) VALUES (25, Temesgen, 'Tadesse');
Query OK, 1 row affected (0.00 sec)
```

```
mysql>
```

Copy and paste the following example in JDBCExample.java, compile and run as follows –

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class JDBCExample {
    static final String DB_URL = "jdbc:mysql://localhost/wsus";
    static final String USER = "thanks";
    static final String PASS = "123";
    static final String QUERY = "SELECT id, first, last, age FROM Employees";
    public static void printResultSet(ResultSet rs) throws SQLException{
        // Ensure we start with first row
        rs.beforeFirst();
        while(rs.next()){
            // Display values
            System.out.print("ID: " + rs.getInt("id"));
            System.out.print(", Age: " + rs.getInt("age"));
            System.out.print(", First: " + rs.getString("first"));
            System.out.println(", Last: " + rs.getString("last"));
        }
        System.out.println();
    }
    public static void main(String[] args) {
        // Open a connection
        try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
            Statement stmt = conn.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
```

```
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery(QUERY);
) {
    System.out.println("List result set for reference....");
    printResultSet(rs);

    // Loop through result set and add 5 in age
    // Move to Before first position so while-loop works properly
    rs.beforeFirst();
    //STEP 7: Extract data from result set
    while(rs.next()){
        // Retrieve by column name
        int newAge = rs.getInt("age") + 5;
        rs.updateDouble( "age", newAge );
        rs.updateRow();
    }
    System.out.println("List result set showing new ages...");
    printResultSet(rs);
    // Insert a record into the table.
    // Move to insert row and add column data with updateXXX()
    System.out.println("Inserting a new record...");
    rs.moveToInsertRow();
    rs.updateString("first","John");
    rs.updateString("last","Paul");
    rs.updateInt("age",40);
    // Commit row
    rs.insertRow();
    System.out.println("List result set showing new set...");
    printResultSet(rs);
} catch (SQLException e) {
    e.printStackTrace();
}
}
```

Now let us compile the above example as follows –

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result –

C:\>java JDBCExample

List result set for reference....

ID: 1, Age: 18, First: Dawit, Last: Uta

ID: 2, Age: 25, First: Temesgen, Last: Tadesse

List result set showing new ages...

ID: 1, Age: 18, First: Dawit, Last: Uta

ID: 2, Age: 25, First: Temesgen, Last: Tadesse

Inserting a new record...

List result set showing new set...

ID: 1, Age: 18, First: Dawit, Last: Uta

ID: 2, Age: 25, First: Temesgen, Last: Tadesse

ID: 5, Age: 40, First: John, Last: Paul

Chapter 8: Servlets

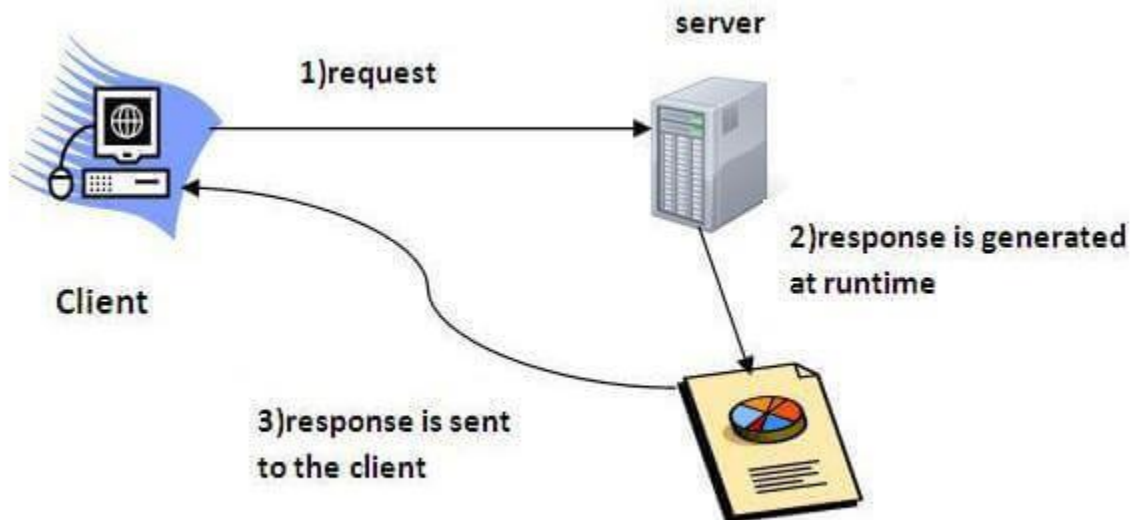
8.1 Introduction to servlets

Servlet technology is used to create a web application (resides at server side and generates a dynamic web page). It is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.



8.2 The Servlet API

Two packages contain the classes and interfaces that are required to build the servlets described in this chapter. These are **javax.servlet** and **javax.servlet.http**. They constitute the core of the Servlet API. Keep in mind that these packages are not part of the Java core packages. Therefore, they are not included with Java SE. Instead, they are provided by Tomcat. They are also provided by Java EE.

The **javax.servlet** package contains a number of interfaces and classes that establish the framework in which servlets operate. The following table summarizes several key interfaces that are provided in this

package. The most significant of these is **Servlet**. All servlets must implement this interface or extend a class that implements the interface. The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

The following table summarizes the core classes that are provided in the **javax.servlet** package:

Class	Description
GenericServlet	Implements the Servlet and ServletConfig interfaces.
ServletInputStream	Encapsulates an input stream for reading requests from a client.
ServletOutputStream	Encapsulates an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

All servlets must implement the **Servlet** interface. It declares the **init()**, **service()**, and **destroy()** methods that are called by the server during the life cycle of a servlet. A method is also provided that allows a servlet to obtain any initialization parameters.

The **init()**, **service()**, and **destroy()** methods are the life cycle methods of the servlet. These are invoked by the server. The **getServletConfig()** method is called by the servlet to obtain initialization parameters. A servlet developer overrides the **getServletInfo()** method to provide a string with useful information (for example, the version number). This method is also invoked by the server.

Method	Description
void destroy()	Called when the servlet is unloaded.
ServletConfig getServletConfig()	Returns a ServletConfig object that contains any initialization parameters.

String getServletInfo()	Returns a string describing the servlet.
void init(ServletConfig <i>sc</i>) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from <i>sc</i> . A ServletException should be thrown if the servlet cannot be initialized.
Void service(ServletRequest <i>req</i> , ServletResponse <i>res</i>) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from <i>req</i> . The response to the client can be written to <i>res</i> . An exception is generated if a servlet or IO problem occurs.

Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **PostParameters.html**, and a servlet is defined in **PostParametersServlet.java**. The HTML source code for **PostParameters.html** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>
<body>
<center>
<form name="Form1" method="post" action="http://localhost:8080/examples/servlets/
servlet/PostParametersServlet">
<table>
<tr>
<td><B>Employee</td>
<td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
<td><B>Phone</td>
<td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

client. The parameter value is obtained via the **getParameter()** method.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response) throws
        ServletException, IOException {
        // Get print writer.
        PrintWriter pw = response.getWriter();
        // Get enumeration of parameter names.
        Enumeration e = request.getParameterNames();
        // Display parameter names and values.
        while(e.hasMoreElements()) {
            String pname = (String)e.nextElement();
            pw.print(pname + " = ");
            String pvalue = request.getParameter(pname);
            pw.println(pvalue);
        }
        pw.close();
    }
}
```

8.3 Request redirecting

The **sendRedirect()** method of **HttpServletResponse** interface can be used to redirect response to another resource, it may be servlet, jsp or html file. It accepts relative as well as absolute URL. It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server.

Difference between forward() and sendRedirect() method

There are many differences between the forward() method of RequestDispatcher and sendRedirect() method of HttpServletResponse interface. They are given below:

forward() method	sendRedirect() method
------------------	-----------------------

The forward() method works at server side.	The sendRedirect() method works at client side.
It sends the same request and response objects to another servlet.	It always sends a new request.
It can work within the server only.	It can be used within and outside the server.
Example: request.getRequestDispatcher("servlet2").forward(request,response);	Example: response.sendRedirect("servlet2");

Syntax of sendRedirect() method

public void sendRedirect(String URL)**throws** IOException;

Example of sendRedirect() method

response.sendRedirect("http://www.javatpoint.com");

Full example of sendRedirect method in servlet

In this example, we are redirecting the request to the google server. Notice that sendRedirect method works at client side, that is why we can our request to anywhere. We can send our request within and outside the server.

DemoServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DemoServlet extends HttpServlet{
    public void doGet(HttpServletRequest req,HttpServletResponse res)
        throws ServletException,IOException
    {
        res.setContentType("text/html");
        PrintWriter pw=res.getWriter();
        response.sendRedirect("http://www.google.com");
        pw.close();
    }
}
```

Creating custom google search using sendRedirect

In this example, we are using sendRedirect method to send request to google server with the request data.*index.html*

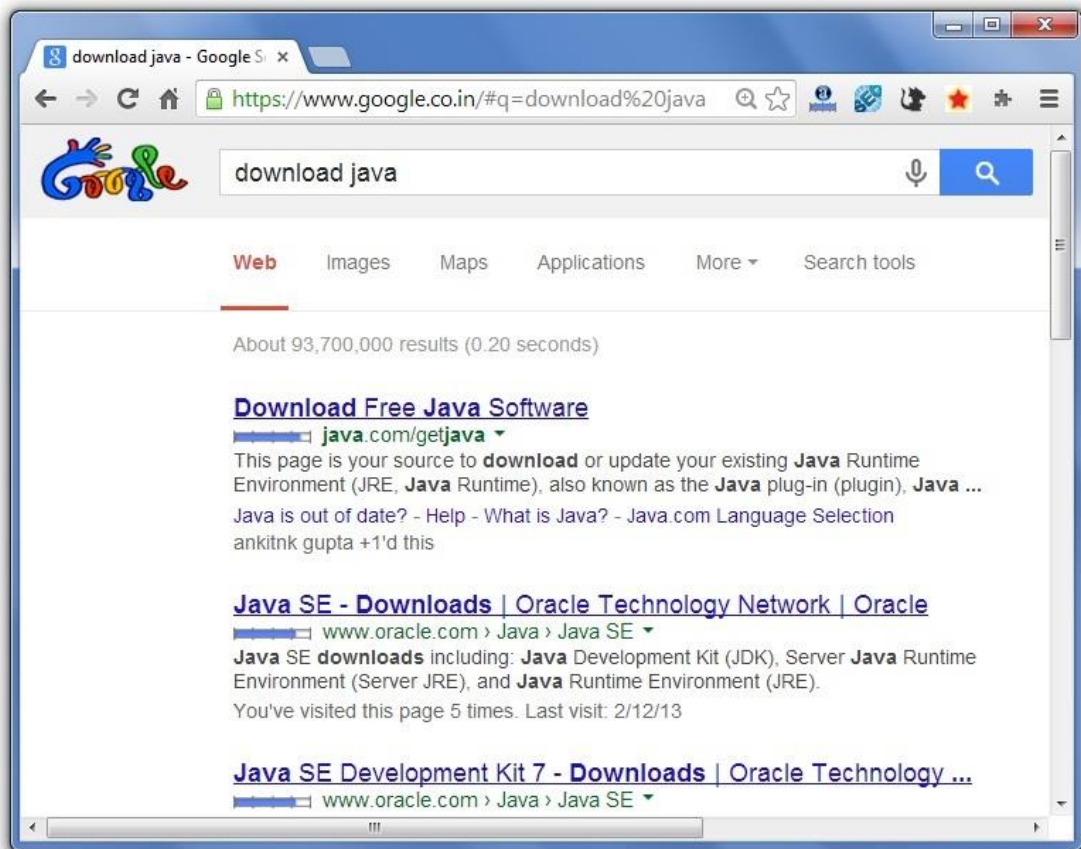
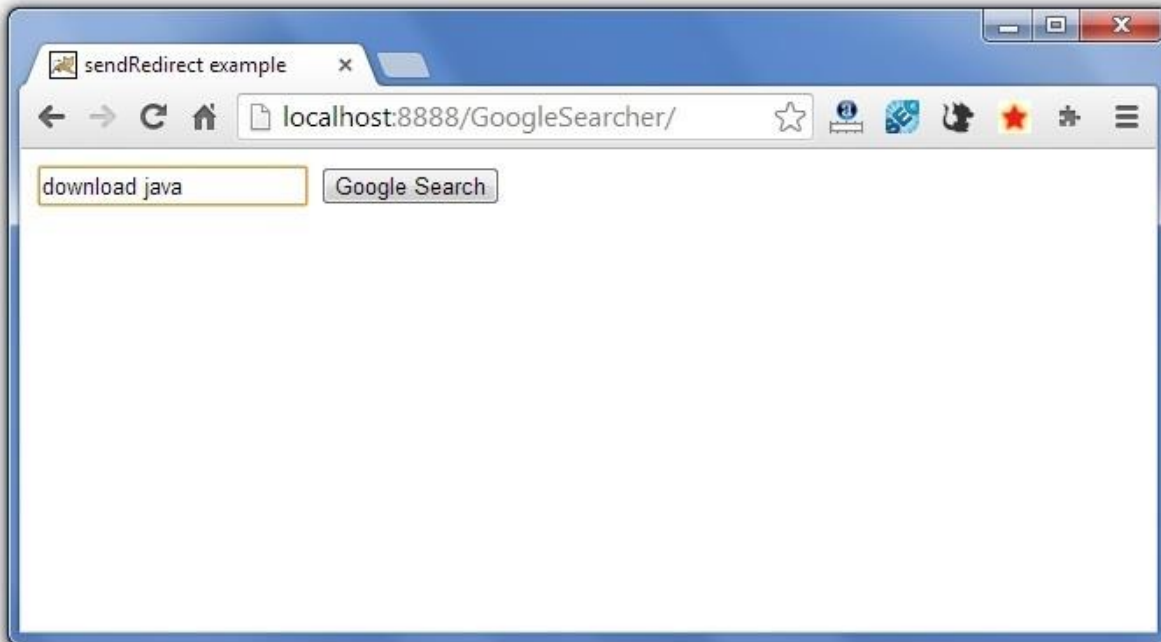
```
<!DOCTYPE html>
<html>
<head>
```

```
<meta charset="ISO-8859-1">
<title>sendRedirect example</title>
</head>
<body>
<form action="MySearcher">
<input type="text" name="name">
<input type="submit" value="Google Search">
</form>
</body>
</html>
```

MySearcher.java

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class MySearcher extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String name=request.getParameter("name");
        response.sendRedirect("https://www.google.co.in/#q="+name);
    }
}
```

Output

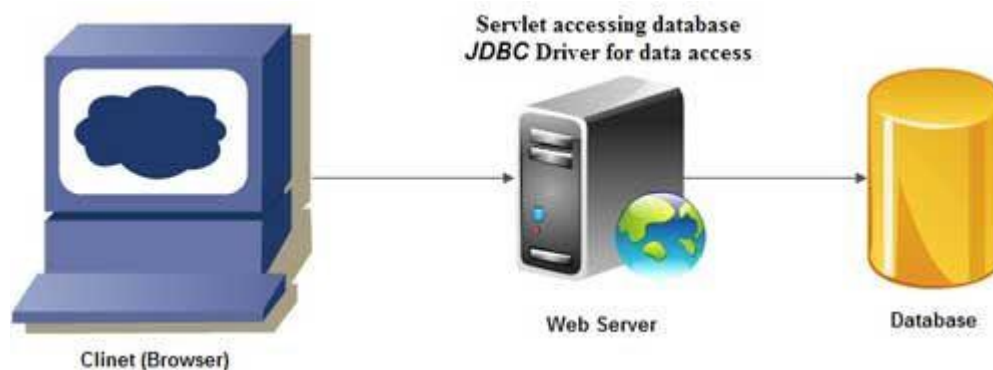


8.4 Multi-tier applications using JDBC from servlet

In [software engineering](#), **multitier architecture** (often referred to as ***n*-tier architecture**) is a [client-server architecture](#) in which presentation, application processing and data management functions are physically separated. The most widespread use of multitier architecture is the **three-tier architecture**.

N-tier application architecture provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific tier, instead of reworking the entire application. A three-tier architecture is typically composed of a *presentation* tier, a [logic](#) tier, and a [data](#) tier.

Typical three-tier architecture for [database](#) applications is shown in the Figure. The first tier (client) contains a browser (such as Netscape or Microsoft IE). The second tier contains the [Java](#)-enabled web server and the [JDBC](#) driver (for example, [JDBC-ODBC](#) bridge driver). The database is contained in the third tier. Servlets allow dynamic HTML pages to communicate with the client and database through JDBC.



To start with interfacing Java Servlet Program with JDBC Connection:

1. Proper JDBC Environment should set-up along with database creation.
2. To do so, download the mysql-connector.jar file from the internet,
3. As it is downloaded, move the jar file to the apache-tomcat server folder,
4. Place the file in lib folder present in the apache-tomcat directory.
5. To start with the basic concept of interfacing:

Step 1: Creation of Database and Table in MySQL

As soon as jar file is placed in the folder, create a database and table in MySQL,

1. `mysql> create database wsu;`
2. `mysql> use wsu`
3. `mysql> create table employee(id int(10), string varchar(20));`

Step 2: Implementation of required Web-pages

Create a form in HTML file, where take all the inputs required to insert data into the database. Specify the servlet name in it, with the POST method as security is important aspects in database connectivity.

```
<!DOCTYPE html>
<html>
<head>
<title>Insert Data</title>
</head>
<body>
  <form action="/InsertData" method="post">
    <p>ID:</p>
    <input type="text" name="id"/> <br/>
    <p>String:</p>
    <input type="text" name="string"/>
    <br/><br/><br/>
    <input type="submit"/>
  </form>
</body>
</html>
```

Step 3: Creation of Java Servlet program with JDBC Connection

To create a JDBC Connection steps are

1. Import all the packages
2. Register the JDBC Driver
3. Open a connection
4. Execute the query, and retrieve the result
5. Clean up the JDBC Environment

Create a separate class to create a connection of database, as it is a lame process to writing the same code snippet in all the program. Create a .java file which returns a Connection object.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
// This class can be used to initialize the database connection
public class DatabaseConnection {
  protected static Connection initializeDatabase()
    throws SQLException, ClassNotFoundException
  {
    // Initialize all the information regarding
    // Database Connection
    String dbDriver = "com.mysql.jdbc.Driver";
```

```
String dbURL = "jdbc:mysql://localhost:3306/";
// Database name to access
String dbName = "demoprj";
String dbUsername = "root";
String dbPassword = "root";
Class.forName(dbDriver);
Connection con = DriverManager.getConnection(dbURL + dbName, dbUsername,
dbPassword);
    return con;
}
}
```

Step 4: To use this class method, create an object in Java Servlet program

Below program shows Servlet Class which create a connection and insert the data in the **demo** table,

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.PreparedStatement;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
// Import Database Connection Class file
import code.DatabaseConnection;
// Servlet Name
@WebServlet("/InsertData")
public class InsertData extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        try {
            // Initialize the database
            Connection con = DatabaseConnection.initializeDatabase();
            // Create a SQL query to insert data into demo table
            // demo table consists of two columns, so two '?' is used
            PreparedStatement st = con.prepareStatement("insert into demo values(?, ?)");
```

```
// For the first parameter,
// get the data using request object
// sets the data to st pointer
st.setInt(1,Integer.valueOf(request.getParameter("id")));
// Same for second parameter
st.setString(2, request.getParameter("string"));
// Execute the insert command using executeUpdate()
// to make changes in database
st.executeUpdate();
// Close all the connections
st.close();
con.close();
// Get a writer pointer
// to display the successful result
PrintWriter out = response.getWriter();
out.println("<html><body><b>Successfully Inserted"
            + "</b></body></html>");
}
catch (Exception e) {
    e.printStackTrace();
}
}
```

Step 5: Get the data from the HTML file

To get the data from the HTML file, the request object is used which calls [getParameter\(\)](#) Method to fetch the data from the channel. After successful insertion, the writer object is created to display a success message. After insertion operation from Servlet, data will be reflected in MySQL Database

References

1. H.M. Deitel, P.J. Deitel, Java How to Program. 8th ed. Prentice Hall
2. Eckel, Bruce. Thinking in Java. 4th Ed. New Jersey: Prentice Hall
3. Harvey M. Deitel and Paul J. Deitel, Java How to Program, Deitel & Associates Inc.
java.sun.com/docs/books/tutorial
4. Tutorialspoint.com