Programming in Java

for

Bachelor of Engineering in Information Technology (BEIT) - IV Semester

AND

Bachelor of Engineering in Software Engineering (BESE) - III Semester

POKHARA UNIVERSITY

Compiled By

Assistant Professor Madan Kadariya

HoD, Department of Information Technology Engineering

NEPAL COLLEGE OF INFORMATION TECHNOLOGY (NCIT)
BALKUMARI, LALITPUR, NEPAL

Table of Contents

Chapter 1: Elements of Java Languages

1.1. History of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. This language was initially called “Oak,” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier and more cost-efficient solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java’s large-scale success.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the “Internet version of C++.” However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come. It is said that computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. However, Java also embodies changes in the way that people approach the writing of programs. For example, Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access. Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe. Java was to Internet programming what C was to system programming: a revolutionary force that changed the world.

Intermediate language, byte-code and code

The programs are first translated into an intermediate language that is the same for all appliances (or all computers), and then a small, easy-to-write and hence, inexpensive program translates this intermediate language into the machine language for a particular appliance or computer. This intermediate language is called Java **bytecode**, or simply, **byte-code**. Since there is only one intermediate language, the hardest step of the two-step translation from program to intermediate language to machine language is the same for all appliances (or all computers); hence, most of the cost of translating to multiple machine languages was saved. The language for programming appliances never caught on with appliance manufacturers, but the Java language into which it evolved has become a widely used programming language. Today, Java is
owned by Oracle Corporation, which purchased Sun Microsystems in 2010.

Why call it byte-code? The word code is commonly used to mean a program or part of a program. A byte is a small unit of storage (eight bits to be precise). Computer readable information is typically organized into bytes. So the term byte-code suggests a program that is readable by a computer as opposed to a person.

1.2. Benefits of Java

1.2.1. Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that someone have some programming experience, he/she will not find Java hard to master. If someone already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if he/she is an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object oriented features of C++, most programmers have little trouble learning Java.

1.2.2. Object-Oriented

Java is an object-oriented programming (OOP) language. The world around us is made up of objects, such as people, automobiles, buildings, streets, adding machines, papers, and so forth. Each of these objects has the ability to perform certain actions, and each of these actions has some effect on some of the other objects in the world. OOP is a programming methodology that views a program as similarly consisting of objects that interact with each other by means of **actions**. Object-oriented programming has its own specialized terminology. The objects are called, appropriately enough, **objects**. The actions that an object can take are called **methods**. Objects of the same kind are said to have the same type or, more often, are said to be in the same class. For example, in an airport simulation program, all the simulated airplanes might belong to the same class, probably called the **Airplane** class. All objects within a class have the same methods. Thus, in a simulation program, all airplanes have the same methods (or possible actions), such as taking off, flying to a specific location, landing, and so forth. However, all simulated airplanes are not identical. They can have different characteristics, which are indicated in the program by associating different data (that is, some different information) with each particular airplane object. For example, the data associated with an airplane object might be two numbers for its speed and altitude.

Things that are called *procedures, methods, functions, or subprograms* in other languages are all called methods in Java. In Java, all methods are part of a class. A Java application program is a class with a method named *main* ; when we run the Java program, the run-time system automatically invokes the method named *main* (that is, it automatically initiates the main action). An application program is a “regular” Java program.

1.2.3. Robust

Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks our code at compile time. However, it also checks our code at run time. Many hard-to track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what we have written will behave in a predictable way under diverse conditions is a key feature of Java.

1.2.4. Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows us to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables us to construct smoothly running interactive systems. Java’s easy-to-use approach to multithreading allows us to think about the specific behaviour of our program, not the multitasking subsystem.

1.2.5. Architecture-Neutral

The compiler generates an architecture-neutral object file format, the compiled code is executable on many processors, given the presence of the Java runtime system. The Java compiler does this by generating ***bytecode*** instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly. Java promised **Write Once, Run Anywhere** (**WORA**), providing no-cost run-times on popular platforms.

1.2.6. Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java ***bytecode***. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. The Java ***bytecode*** was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a ***just-in-time*** compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

1.2.7. Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports ***Remote Method Invocation (RMI)***. This feature enables a program to invoke methods across a network.

1.2.8. Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of *bytecode* may be dynamically updated on a running system.

1.2.9. Secure

Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems. Java was designed to make certain kinds of attacks impossible, among them:

- Overrunning the runtime stack—a common attack of worms and viruses
- Corrupting memory outside its own process space
- Reading or writing files without permission

1.2.10. Portable

Unlike C and C++, there are no "implementation-dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behaviour of arithmetic on them.

A First Simple Program

```
/* This is a simple Java program. Call this file "Example.java". */
class Example {
// program begins with a call to main().
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

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

- **Case Sensitivity -** Java is case sensitive which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names -** For all class names the first letter should be in Upper Case. If several words are used to form a name of the class each inner words first letter should be in Upper Case. Example *class MyFirstJavaClass*
- **Method Names -** All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
  Example *public void myMethodName()*
- **Program File Name -** Name of the program file should exactly match the public class name. When saving the file we should save it using the class name and append '.java' to the end of the name. (if the file name and the class name do not match then the program will not compile).
  Example : Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*
- **public static void main(String args[]) -** java program processing starts from the main() method which is a mandatory part of every java program.

A Closer Look at the First Sample Program
Line 1: Program starts with comment. For multiline comment we can use comment inside /* …..*/. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code.
Line 2: This line uses the keyword ***class*** to declare that a new class is being defined. **Example** is an ***identifier*** that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).
Line 3: The next line in the program is the single-line comment.

Line 4: This line begins the **main ()** method. This is the line at which the program will begin executing. All Java applications begin execution by calling **main ()**. The **public** keyword is an **access specifier**, which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by *code outside the class* in which it is declared. (The opposite of public is private, which prevents a member from being used by code defined outside of its class.) In this case, **main ()** must be declared as public, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main ()** to be called without having to *instantiate a particular instance of the class.* This is necessary since main () is called by the Java Virtual Machine before any objects are made. The keyword **void** simply tells the compiler that **main ()** does not return a value.
String args[ ] declares a parameter named **args**, which is an array of instances of the class String. Objects of type String store character strings. In this case, **args** receives any *command-line arguments* present when the program is executed. This program does not make use of this information, but other programs may have information of it.The last character on the line is the {. This signals the start of **main ()**'s body. One other point: **main ()** is simply a starting place for our program. A complex program will have dozens of classes, only one of which will need to have a **main ()** method to get things started.
Line 5: This line outputs the string "*This is a simple Java program.*" followed by a new line on the screen. Output is actually accomplished by the built-in **println()** method. In this case, **println ()** displays the string which is passed to it. **println( )** can be used to display other types of information, too. The line begins with **System.out**. **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.
Line 6 and 7: The first} in the program ends **main()**, and the last } ends the Example class definition

Syntax and Semantics
The description of a programming language, or any other kind of language, can be thought of as having two parts, called the **syntax** and **semantics** of the language. The **syntax** tells what arrangement of words and punctuation is legal in the language. The **syntax** is often called the language's grammar rules. For Java, the syntax describes what arrangements of words and punctuation are allowed in a class or program definition. The **semantics** of a language describes the meaning of things written while following the syntax rules of the language. For a Java program, the syntax describes how to write a program and the semantics describes what happens when we run the program. When writing a program in Java, we are always using both the syntax and the semantics of the Java language.

1.3. Data Types

Java Is a Strongly Typed Language
It is important to state at the outset that Java is a strongly typed language. First, in java every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

1.3.1. The Primitive Types

Java defines **eight** primitive types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as simple types. The primitive types represent single values, not complex objects. Although Java is otherwise completely object-oriented, the primitive types are not. They are analogous to the simple types found in most other non object oriented languages. The reason for this is efficiency. Making the primitive types into objects would have degraded performance too much.

1. **Integers**: This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. The concept of unsigned was used mostly to specify the behavior of the high-order bit, which defines the sign of an integer value. The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Here is a program that computes the number of miles that light will travel in a specified number of days.

```
// Compute distance light travels using long variables.
class Light {
     public static void main(String args[]) {
          int lightspeed;
          long days;
          long seconds;
          long distance;
          // approximate speed of light in miles per second
          lightspeed = 186000;
          days = 1000; // specify number of days here
          seconds = days * 24 * 60 * 60; // convert to seconds
          distance = lightspeed * seconds; // compute distance
          System.out.print("In " + days);
          System.out.print(" days light will travel about ");
          System.out.println(distance + " miles.");
     }
}
```

Output:
```
In 1000 days light will travel about 16070400000000 miles.
```

1. **Floating-point numbers**: This group includes **float** and **double**, which represent numbers with fractional precision. Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as **sine** and **cosine**, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Here is a short program that uses double variables to compute the area of a circle:

```
// Compute the area of a circle.
```

```
class Area {
     public static void main (String args []) {
          double pi, r, a;
          r = 10.8; // radius of circle
          pi = 3.1416; // pi, approximately
          a = pi * r * r; // compute area
          System.out.println("Area of circle is " + a);
     }
}
```

2. **Characters:** This group includes **char**, which represents symbols in a character set, like letters and numbers. **Char** in Java is not the same as char in C or C++. In C/C++, **char** is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters. Unicode is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java char is a *16-bit* type. The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Here is a program that demonstrates char variables:

```
// Demonstrate char data type.
class CharDemo {
     public static void main (String args[]) {
          char ch1, ch2;
          ch1 = 88; // code for X
          ch2 = 'Y';
          System.out.print("ch1 and ch2: ");
          System.out.println(ch1 + " " + ch2);
     }
}
```

Output: `ch1 and ch2: X Y`

3. **Boolean**: This group includes **boolean**, which is a special type for representing **true/false** values. This is the type returned by all relational operators, as in the case of a < b. **boolean** is also the type required by the conditional expressions that govern the control statements such as if and for. Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
       boolean b;
       b = false;
       System.out.println("b is " + b);
       b = true;
       System.out.println("b is " + b);
       // a boolean value can control the if statement
       if(b)
         System.out.println("This is executed.");
       b = false;
       if(b)
            System.out.println("This is not executed.");
       // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

Output:

```
b is false
b is true
This is executed.
   10  9 is true
```

1.3.2. Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

1.4. Java Variables

1.4.1. Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

```
public class Test{
   public void pupAge(){
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

This would produce following result:

```
Puppy age is: 7
```

Example:

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test{
   public void pupAge(){
      int age;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }
   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

This would produce following error while compiling it:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
        ^
   2. error
```

1.4.2. Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present through out the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods and different class (when instance variables are given accessibility) the should be called using the fully qualified name.

  ObjectReference.VariableName.

Example:

```
import java.io.*;
public class Employee {
   // this instance variable is visible for any child class.
   public String name;
   // salary variable is visible in Employee class only.
   private double salary;
   // The name variable is assigned in the constructor.
   public Employee (String empName){
      name = empName;
   }
   // The salary variable is assigned a value.
   public void setSalary(double empSal){
      salary = empSal;
   }
   // This method prints the employee details.
   public void printEmp(){
      System.out.println("name  : " + name );
      System.out.println("salary :" + salary);
   }
   public static void main(String args[]){
      Employee empOne = new Employee("Ransika");
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

Output:

```
name  : Ransika
salary :1000.0
```

1.4.3. Class/static variables:

- Class variables also known as static variables are declared with the ***static*** keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name. *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Example:

```
import java.io.*;
public class Employee{
   // salary  variable is a private static variable
   private static double salary;
   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";
   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT+"average salary:"+salary);
   }
}
```

Output:

```
Development average salary:1000
```

Note: If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

1.5. Java Identifiers

The name of something in a Java program, such as a variable, class, method, or object name, must not start with a digit and may only contain letters, digits (0 through 9), and the underscore character ( ). Upper- and lowercase letters are considered to be different characters. (The symbol $ is also allowed, but it is reserved for special purposes only, so we should not typically use $ in a Java name.) Names in a program are called identifiers.
In java there are several points to remember about identifiers. They are as follows:

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly identifiers are case sensitive.
- Examples of legal identifiers: age, $salary, _value, __1_value
- Examples of illegal identifiers: 123abc, -salary

1.6. Java Enums

- **Enums** were introduced in java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.
- With the use of **enums** it is possible to reduce the number of bugs in your code.
- For example if we consider an application for a fresh juice shop it would be possible to restrict the glass size to small, medium and Large. This would make sure that it would not allow anyone to order any size other than the small, medium or large.

Example

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

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

```
    }
}
```
Output:
```
Size: MEDUIM
```

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

1.7. Java Keywords

There is a special class of identifiers, called keywords or reserved words, that have a predefined meaning in Java and that you cannot use as names for variables or anything else. The following list shows the reserved words in Java.

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

Accessing Instance Variables and Methods
Instance variables and methods are accessed via created objects. To access an instance variable, the fully qualified path should be as follows:
```
/* First create an object */
ObjectReference = new Constructor();
/* Now call a variable as follows */
ObjectReference.variableName;
/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example
This example explains how to access instance variables and methods of a class:
```
public class Puppy{
   int puppyAge;
   public Puppy(String name){
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }
   public void setAge( int age ){
       puppyAge = age;
   }
   public int getAge( ){
       System.out.println("Puppy's age is :" + puppyAge );
       return puppyAge;
   }
   public static void main(String []args){
      /* Object creation */
      Puppy myPuppy = new Puppy( "tommy" );
      /* Call class method to set puppy's age */
      myPuppy.setAge( 2 );

      /* Call another class method to get puppy's age */
      myPuppy.getAge( );
      /* You can access instance variable as follows as well */
      System.out.println("Variable Value :" + myPuppy.puppyAge );
   }
}
```

Output:

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

Source file declaration rules
These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only *one public class per source file*.
- A source file can have *multiple non public classes*.
- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example: The class name is . *public class Employee {}* Then the source file should be as **Employee.java**.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes etc.Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

A Simple Case Study:
For our case study we will be creating two classes. They are **Employee** and **EmployeeTest**. The Employee class has four instance variables name, age, designation and salary. The class has one explicitly defined constructor which takes a parameter.

File: Employee.Java

```
import java.io.*;
public class Employee{
   String name;
   int age;
   String designation;
   double salary;
   // This is the constructor of the class Employee
   public Employee(String name){
      this.name = name;
   }
   // Assign the age of the Employee to the variable age.
   public void empAge(int empAge){
      age =  empAge;
   }
   /* Assign the designation to the variable designation.*/
   public void empDesignation(String empDesig){
      designation = empDesig;
   }
   /* Assign the salary to the variable   salary.*/
   public void empSalary(double empSalary){
      salary = empSalary;
   }
   /* Print the Employee details */
   public void printEmployee(){
      System.out.println("Name:"+ name);
      System.out.println("Age:" + age);
      System.out.println("Designation:" + designation);
      System.out.println("Salary:" + salary);
   }
}
```

In-order to run this Employee class there should be main method and objects should be created. We will be creating a separate class for these tasks.

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

File: EmployeeTest.java

```
import java.io.*;
public class EmployeeTest{
   public static void main (String args[]){
      /* Create two objects using constructor */
      Employee empOne = new Employee ("James Smith");
      Employee empTwo = new Employee ("Mary Anne");
      // Invoking methods for each object created
      empOne.empAge(26);
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

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

1.8. Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10),hexadecimal(base 16) or octal(base 8) number systems as well.
Prefix 0 is used to indicates octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

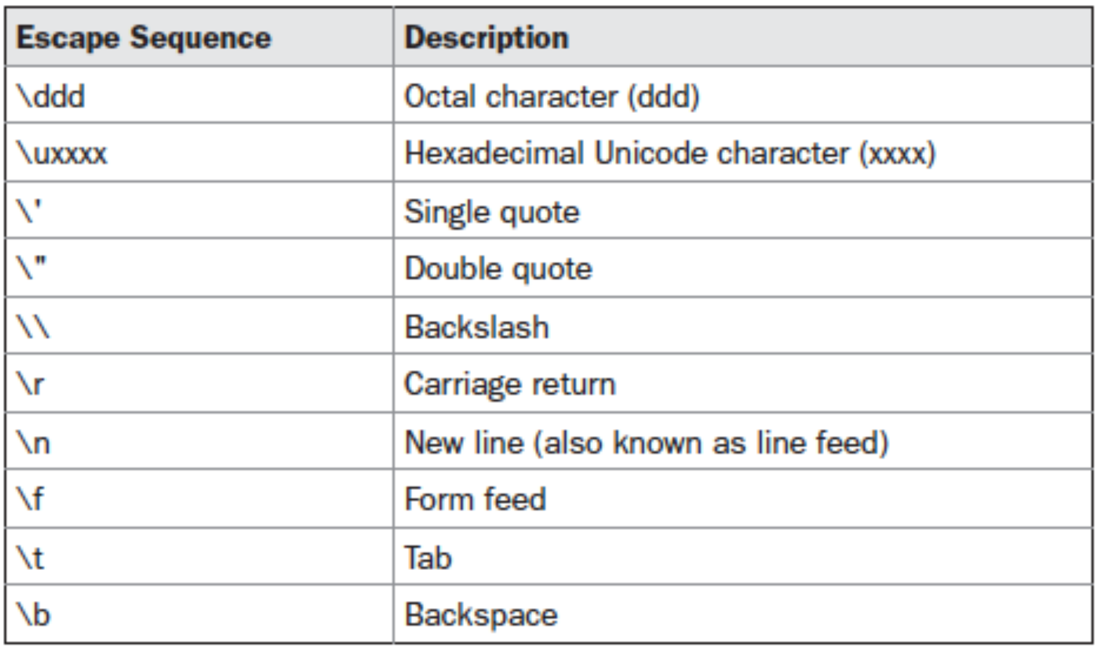
String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:



Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages.

1.9. Dynamic Initialization of Variables

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class DynInit {
      public static void main(String args[]) {
            double a = 3.0, b = 4.0;
            // c is dynamically initialized
            double c = Math.sqrt(a * a + b * b);
            System.out.println("Hypotenuse is " + c);
      }
}
```

Here, three local variables: a, b, and c are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's built-in methods, sqrt( ), which is a member of the **Math** class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

1.10.The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the **main**( ) method. However, Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time we start a new block, we are creating a new scope. A scope determines what objects are visible to other parts of our program. It also determines the lifetime of those objects.

Many other computer languages define two general categories of scopes: **global** and **local**. However, these traditional scopes do not fit well with Java's strict, object-oriented model. While it is possible to create what amounts to being a **global** scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense.

For now, we will only examine the scopes defined by or within a method. The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when we declare a variable within a scope, we are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the *foundation for encapsulation*. Scopes can be nested. For example, each time we create a block of code, we are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.
To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
```

```
class Scope {
     public static void main(String args[]) {
          int x; // known to all code within main
          x = 10;
          if(x == 10) { // start new scope
               int y = 20; // known only to this block
               // x and y both known here.
               System.out.println("x and y: " + x + " " + y);
               x = y * 2;
          }
     // y = 100; // Error! y not known here
     // x is still known here.
     System.out.println("x is " + x);
     }
}
```

1.11.Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

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

1.11.1. The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
++	Increment - Increase the value of operand by 1
--	Decrement - Decrease the value of operand by 1

1.11.2. The Relational Operators

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example

==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

1.11.3.The Bitwise Operators

Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte. Bitwise operator works on bits and perform bit by bit operation.
Assume if a = 60; and b = 13; Now in binary format they will be as follows:
a = 0011 1100
b = 0000 1101
----
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011
The following table lists the bitwise operators:Assume integer variable A holds 60 and variable B holds 13 then

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000

>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

1.11.4.The Logical Operators

The following table lists the logical operators: Assume boolean variables A holds true and variable B holds false then

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

1.11.5. The Assignment Operators

There are following assignment operators supported by Java language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2

&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

1.11.6.Misc Operators

There are few other operators supported by Java Language.
Conditional Operator (? :):
Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as :
`variable x = (expression) ? value if true : value if false`
Example:

```
public class Test {
   public static void main(String args[]){
      int a , b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );
      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

Output:

```
Value of b is : 30
Value of b is : 20
```

1.11.7.instanceOf Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). **instanceOf** operator is wriiten as:
`( Object reference variable ) instanceOf  (class/interface type)`
If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side then the result will be true. Following is the example:

```
String name = = 'James';
boolean result = name instanceOf String;
// This will return true since name is type of String
```

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle {}
public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result);
   }
}
```
Output:
```
True
```

1.12. Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:
For example, x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedenace than + so it first gets multiplied with 3*2 and then adds into 7.
Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Table 1: Operator Precedence

1.13. Iteration Statements in Java

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops.

1.13.1. The while Loop

A while loop is a control structure that allows us to repeat a task a certain number of times.
Syntax:
The syntax of a while loop is:
```
while(Boolean_expression)
{
   //Statements
}
```

When executing, if the *boolean_expression* result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true. Here key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example
```
public class Test {
   public static void main(String args[]) {
      int x = 10;
      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }
   }
}
```
Output:
```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

1.13.2. The do...while Loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax
```
do
{
   //Statements
}while(Boolean_expression);
```
Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested. If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:
```
public class Test {
   public static void main(String args[]){
      int x = 10;
      do{
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }while( x < 20 );
   }
}
```

Output:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

1.13.3 The for Loop

A for loop is a repetition control structure that allows us to efficiently write a loop that needs to execute a specific number of times.A for loop is useful when we know how many times a task is to be repeated.

Syntax

```
for(initialization; Boolean_expression; update)
{
   //Statements
}
```

Here is the flow of control in a for loop:

- The initialization step is executed first, and only once. This step allows us to declare and initialize any loop control variables. We are not required to put a statement here, as long as a semicolon appears.
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps backs up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {
   public static void main(String args[]) {
      for(int x = 10; x < 20; x = x+1) {
         System.out.print("value of x : " + x );
         System.out.print("\n");
      }
   }
}
```

Output:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

1.13.4. Enhanced for loop in Java:

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

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

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

Example

```
public class Test {
   public static void main(String args[]){
      int [] numbers = {10, 20, 30, 40, 50};
      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names ={"James", "Larry", "Tom", "Lacy"};
      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

```
Output:
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

1.13.5. The break Keyword:

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

```
break;
```

Example:

```
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};
      for(int x : numbers ) {
         if( x == 30 ) {
             break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

```
Output:

10
20
```

1.13.6. The continue Keyword

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

```
continue;
```

Example:

```
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};
      for(int x : numbers ) {
         if( x == 30 ) {
             continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

```
Output:
10
20
40
50
```

1.14. Decision making statements/selection statements in Java

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time

1.14.1. The if Statement:

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if(Boolean_expression)
{
   //Statements will execute if the Boolean expression is true
}
```

If the boolean expression evaluates to true, then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement (after the closing curly brace) will be executed.

```
public class Test {
   public static void main(String args[]){
      int x = 10;
      if( x < 20 ){
         System.out.print("This is if statement");
      }
   }
}
```

Output:

```
This is if statement
```

1.14.2. The if...else Statement:

An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

```
if(Boolean_expression){
   //Executes when the Boolean expression is true
}else{
   //Executes when the Boolean expression is false
}
```

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 30;
      if( x < 20 ){
         System.out.print("This is if statement");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

Output:

```
This is else statement
```

1.14.3.The if...else if...else Statement:

An if statement can be followed by an optional *else if...else* statement, which is very usefull to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
}else {
   //Executes when the none of the above condition is true.
}
```

Example:

```
public class Test {
   public static void main(String args[]){
      int x = 30;
      if( x == 10 ){
         System.out.print("Value of X is 10");
      }else if( x == 20 ){
         System.out.print("Value of X is 20");
      }else if( x == 30 ){
         System.out.print("Value of X is 30");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

Output

```
Value of X is 30
```

1.14.4. Nested if...else Statement:

It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement.

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
   if(Boolean_expression 2){
      //Executes when the Boolean expression 2 is true
   }
}
```

You can nest *else if...else* in the similar way as we have nested *if* statement.

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 30;
      int y = 10;
      if( x == 30 ){
         if( y == 10 ){
             System.out.print("X = 30 and Y = 10");
          }
       }
    }
}
```

Output

```
X = 30 and Y = 10
```

1.14.5.The switch Statement:

A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number of case statements.
    default : //Optional
       //Statements
}
```

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- We can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {
   public static void main(String args[]){
      //char grade = args[0].charAt(0);
      char grade = 'C';
      switch(grade)
      {
         case 'A' :
            System.out.println("Excellent!");
            break;
         case 'B' :
         case 'C' :
            System.out.println("Well done");
            break;
         case 'D' :
            System.out.println("You passed");
         case 'F' :
            System.out.println("Better try again");
            break;
         default :
            System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

Output:

```
$ java Test
Well done
Your grade is a C
$
```

1.15. Arrays in Java

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

1.15.1. One-Dimensional Arrays

Aone-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is ***type var-name[ ];***

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named month_days with the type "array of int":
int month_days[];
Although this declaration establishes the fact that ***month_days*** is an array variable, no array actually exists. In fact, the value of ***month_days*** is set to null, which represents an array with no value. To link ***month_days*** with an actual, physical array of integers, we must allocate one using new and assign it to month_days. new is a special operator that allocates memory.
array-var = new type[size];
month_days = new int[12];

Hence, Obtaining an array is a two-step process.
1. First, We must declare a variable of the desired array type.
2. Second, we must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

```
// Demonstrate a one-dimensional array.
public class Array {
      public static void main(String args[]){
           int monthDays[];
           monthDays=new int[12];
           monthDays[0]=31;
           monthDays[1]=28;
           monthDays[2]=31;
           monthDays[3]=30;
           monthDays[4]=31;
           monthDays[5]=30;
           monthDays[6]=31;
           monthDays[7]=31;
           monthDays[8]=30;
           monthDays[9]=31;
           monthDays[10]=30;
           monthDays[11]=31;
           System.out.println("January has " +monthDays[0] + " days");
           System.out.println("February has " +monthDays[1] + " days");
           System.out.println("March has " +monthDays[2] + " days");
           System.out.println("April has " +monthDays[3] + " days");
           System.out.println("May has " +monthDays[4] + " days");
           System.out.println("June has " +monthDays[5] + " days");
           System.out.println("July has " +monthDays[6] + " days");
           System.out.println("August has " +monthDays[7] + " days");
           System.out.println("September has " +monthDays[8] + " days");
           System.out.println("October has " +monthDays[9] + " days");
           System.out.println("November has " +monthDays[10] + " days");
           System.out.println("December has " +monthDays[11] + " days");
      }
}

// An improved version of the previous program.
class AutoArray {
      public static void main(String args[]) {
           int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                                       30, 31 };
           System.out.println("January has " +monthDays[0] + " days");
           System.out.println("February has " +monthDays[1] + " days");
           System.out.println("March has " +monthDays[2] + " days");
           System.out.println("April has " +monthDays[3] + " days");
           System.out.println("May has " +monthDays[4] + " days");
           System.out.println("June has " +monthDays[5] + " days");
           System.out.println("July has " +monthDays[6] + " days");
           System.out.println("August has " +monthDays[7] + " days");
           System.out.println("September has " +monthDays[8] + " days");
           System.out.println("October has " +monthDays[9] + " days");
           System.out.println("November has " +monthDays[10] + " days");
           System.out.println("December has " +monthDays[11] + " days");
      }
}
```

Output:

```
January has 31 days
February has 28 days
March has 31 days
April has 30 days
May has 31 days
```

```
June has 30 days
July has 31 days
August has 31 days
September has 30 days
October has 31 days
November has 30 days
December has 31 days
```

It finds the average of a set of numbers.

```
// Average an array of values.
class Average {
     public static void main(String args[]) {
          double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
          double result = 0;
          int i;
          for(i=0; i<5; i++)
               result = result + nums[i];
          System.out.println("Average is " + result / 5);
     }
}
```

1.15.2. Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as we might expect, look and act like regular multidimensional arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a twodimensional array variable called twoD.

int twoD[][] = new int[4][5];

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int. The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
     public static void main(String args[]) {
          int twoD[][]= new int[4][5];
          int i, j, k = 0;
          for(i=0; i<4; i++)
               for(j=0; j<5; j++) {
                    twoD[i][j] = k;
                    k++;
               }
               for(i=0; i<4; i++) {
                    for(j=0; j<5; j++)
                         System.out.print(twoD[i][j] + " ");
                    System.out.println();
          }
     }
}

Output:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

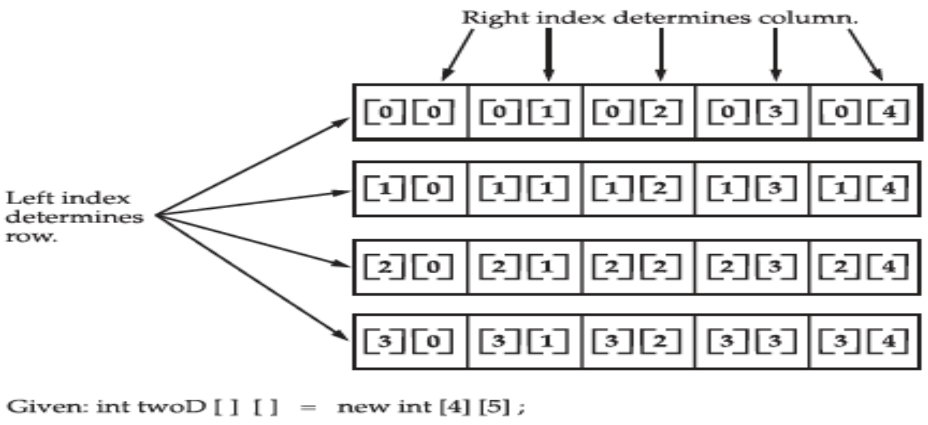


Figure 1: A conceptual view of a 4 by 5, two-dimensional array

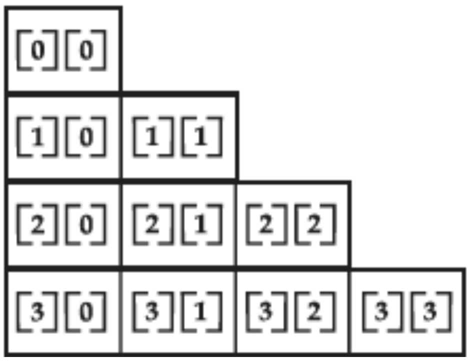
When we allocate memory for a multidimensional array, We need only specify the memory for the first (leftmost) dimension. We can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It allocates the second dimension manually.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];

Example
// Manually allocate differing size second dimensions.
class TwoDAgain {
     public static void main(String args[]) {
          int twoD[][] = new int[4][];
          twoD[0] = new int[1];
          twoD[1] = new int[2];
          twoD[2] = new int[3];
          twoD[3] = new int[4];
          int i, j, k = 0;
          for(i=0; i<4; i++)
               for(j=0; j<i+1; j++) {
                    twoD[i][j] = k;
                    k++;
               }
               for(i=0; i<4; i++) {
                    for(j=0; j<i+1; j++)
                         System.out.print(twoD[i][j] + " ");
                    System.out.println();
                    }
               }
          }
```

Output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



Irregular arrays can be used effectively in some situations. For example, if we need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

1.15.3.Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

```
type[ ] var-name;
```

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
```

creates three array variables of type int. It is the same as writing

```
int nums[], nums2[], nums3[]; // create three arrays
```

The alternative declaration form is also useful when specifying an array as a return type for a method.

1.16. Strings

In java's **String**, is not a simple type. Nor is it simply **an array of characters**. Rather, **String is** defined as an object, and a full description of it requires an understanding of several object-related features.
The **String** type is used to declare string variables. We can also declare arrays of strings. A quoted string constant can be assigned to a **String** variable. A variable of type **String** can be assigned to another variable of type **String**. We can use an object of type **String** as an argument to **println( )**. For example, consider the following fragment:

```
String str = "this is a test";
System.out.println(str);
```

Here, **str** is an object of type **String**. It is assigned the string "this is a test". This string is displayed by the **println( )** statement. **String** objects have many special features and attributes that make them quite powerful and easy to use.

```
public class StringProgram {
     public static void main(String args[]){
          String name = "First Name";
          //String name = new String("First Name ");
          System.out.println("my name is " +name);
}
}
```

Output:

```
my name is First Name
```

Another Program:

```
public class StringProgram {
   public static void main(String args[]){
      String name1 = new String("Mahesh Babu");
      String nameInUpperCase=name1.toUpperCase();
```

```
        //String class contains toUpperCase() method to convert string to
        upper case
        System.out.println("Name in upper case is "+nameInUpperCase);
        String nameInLowerCase=name1.toLowerCase();
        //String class contains tolowerCase() method to convert string to
        lower case
        System.out.println("name in lower case is "+nameInLowerCase);
        String name2 = new String("Kajal Agrawal");
        int result;
        result=name1.compareTo(name2);
        //string class contains compareTo() method to compare two string
        if(result==0)
        System.out.println("the two words are same");
        if(result<0)
        System.out.println(name1+" comes first in dictionary order than
        "+name2);
        if(result>0)
        System.out.println(name2+" comes first in dictionary order
        "+name1);
        char check='@';
        String emailAddress = "exampleemail.com";
        result=emailAddress.indexOf(check);
        //indexof returns position in which the character is found, if not
        found returns -1
        System.out.println(result);
        if(result==-1)
        System.out.println("invalid email address");
    }
}
```

Output:

```
Name in upper case is MAHESH BABU
name in lower case is mahesh babu
Kajal Agrawal comes first in dictionary order Mahesh Babu
-1
invalid email address
```

//additional program to illustrate the concept of String

```
public class StringProgram {
    public static void main(String[] args) {
        String str1 = "This Is Java Programming";
        int lengthOfString = str1.length();//Returns the length of this
string.
        System.out.println("Length of String= " + lengthOfString);
        String str2 = "This Is C Programming";
        String trimStr1 = str1.trim();//Returns a copy of the string, with
leading and trailing whitespace omitted.
        System.out.println(trimStr1);
        boolean b = str1.equals(str2);
        System.out.println("boolean value " + b);
        b = str1.equalsIgnoreCase(str2);
        System.out.println("boolean value " + b);
        String str3 = "";
       if (str3.isEmpty()) {
            System.out.println("string is empty");
        }
    }
}
```

Output:

```
Length of String= 24
This Is Java Programming
boolean value false
boolean value false
string is empty
```

Chapter 2: Object Oriented Programming in Java

2.1. Classes and object

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. The class forms the basis for object-oriented programming in Java. Any concept we wish to implement in a Java program must be encapsulated within a class.

2.1.1. Class Fundamentals

Upto now, the created classes are primarily existing simply to encapsulate the main () method, which has been used to demonstrate the basics of the Java syntax. 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.

2.1.2. The General Form of a Class

For declaring class, we declare its exact form and nature. This can be done 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.

A class is declared by use of the **class** keyword. Classes can (and usually do) get much more complex. A simplified 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, as a general rule, 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.

2.1.3. A Simple Class

Here is a class called **Box** that defines three instance variables: *width*, *height*, and *depth*. Currently,

Box does not contain any methods:
```
      class Box {
            double width;
            double height;
            double depth;
      }
```
As stated, a class defines a new type of data. In this case, the new data type is called **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, we 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. Creation of instance of class means 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*.

Accessing instance variable

To access these variables, *dot* (.) operator is used. 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, we 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, we 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.
public 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);
      }
}
```
File that contains this program should be **BoxDemo.java**, because the **main( )** method is in the class called **BoxDemo**, not the class called **Box**. After compilation there will be 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.

Note: ***It is not necessary for both the Box and the BoxDemo class to actually be in the same source file. We could put each class in its own file, called Box.java and BoxDemo.java, respectively. To run this program, we must execute BoxDemo.class.***

Output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if we 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);
      }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Here, **mybox1**'s data is completely separate from the data contained in **mybox2**.

2.1.4. Declaring Objects

Creating a class means we are creating a new data type. However, **obtaining objects of a class is a two-step process:**

- **First, we 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, we must acquire an actual, physical copy of the object and assign it to that variable. We can do this using the new operator.**

Note: ***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, we 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 in Figure below.

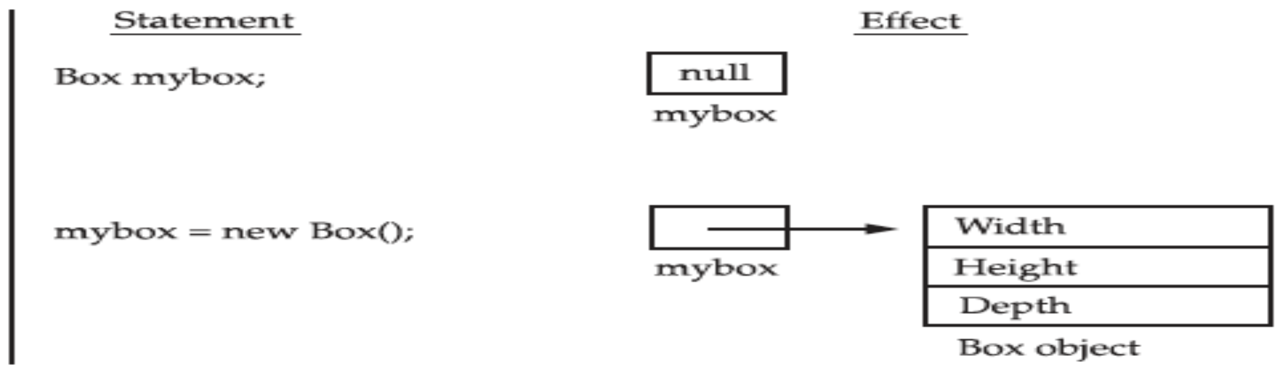


Figure 2: Declaring an object of type Box

2.1.5. A Closer Look at new

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.
Why you do not need to use **new** for such things as integers or characters?
The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "**normal**" variables. Java can implement the primitive types more efficiently. 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 we declare an object of a class, we 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

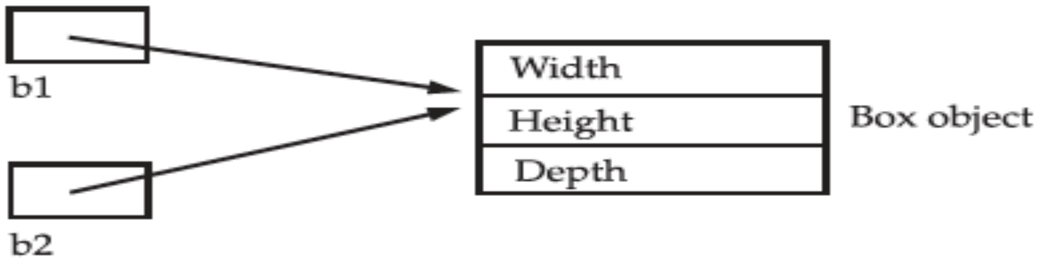
in memory.) It is important to keep this distinction clearly in mind.

2.1.6. Assigning Object Reference Variables

Object reference variables act differently than we might expect when an assignment takes place. For example,

```
Box b1 = new Box();
Box b2 = b1;
```

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 does **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.

Note: When we assign one object reference variable to another object reference variable, we are not creating a copy of the object, we are only making a copy of the reference.

Example

```
class Rectangle {
  double length;
}
class RectangleDemo {
  public static void main(String args[]) {

  Rectangle r1 = new Rectangle();
  Rectangle r2 = r1;
  r1.length = 10;
  r2.length = 20;

  System.out.println("Value of R1's Length : " + r1.length);
  System.out.println("Value of R2's Length : " + r2.length);

  r1=null;
  r2.length = 30;
 //System.out.println("Value of R1's Length 2: " + r1.length); //Exception
  System.out.println("Value of R2's Length 2: " + r2.length);
  }
}
```

Output:
```
Value of R1's Length : 20.0
Value of R2's Length : 20.0
Value of R2's Length 2: 30.0
```

2.1.7. Introducing Methods

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 we need to learn now so that we can begin to add methods to our classes.
This is the general form of a method:
```
Return_type functionName(parameter-list) {
// body of method
}
```

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

2.1.8. Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens. Most of the time, we will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, we can also define methods that are used internally by the class itself.

In 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, the method should add 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();
      }
}
```

Output:

```
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. 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.*

Note: 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.

2.1.9. 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( )**.

2.1.10. 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. 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. 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.
Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions 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);
}
}
```

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.

2.2. 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);
}
}
```

Output:

```
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.
When you allocate an object, you use the following general form:

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

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.
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.

2.2.1. 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. The following version of **Box** defines a parameterized constructor that 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);
}
}
```

Output:

```
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.

2.2.2. 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.

2.2.3. Instance Variable Hiding

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.

2.2.4. 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.

2.2.5. 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 character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.*

By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a ***finalizer*** to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.
The **finalize( )** method has this general form:
```
protected void finalize( )
{
// 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.

2.3. Methods and Classes

2.3.1. 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 supports **polymorphism**. Method overloading is one of Java's most exciting and useful features. 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);
}
}
```

Output:

```
No parameters a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5
```

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.

2.3.2. 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 return to the **Box** class developed in the preceding chapter. 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);
      }
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

The proper overloaded constructor is called based upon the parameters specified when **new** is executed.

2.3.3. Using Objects as Parameters

Until now, 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));
}
}
```

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.

2.3.4. Argument Passing (Call-by-Value and Call-by-Reference)

In general, there are two ways that a computer language can pass an argument to a subroutine: The first way is ***call-by-value**:* This approach 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.

The second way an argument can be passed is ***call-by-reference**:* In this approach, 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.

In Java, when you pass a primitive 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:

```
// Primitive 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);
     }
 }
```

Output::

```
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 what is effectively call-by-reference. When you pass the 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 own classes, providing many forms of constructors is usually required to allow objects to be constructed in a convenient and efficient manner.

```
//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);
 }
}
```

output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

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

2.3.5. Returning Objects

A method can return any type of data, including class types. 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);
       }
```

}
Output:

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

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**, it is not 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 program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

2.3.6. Access specifier

Java's access specifiers are

1. **public**,
2. **private**, and
3. **protected**.

Note: Java also defines a default access level. Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public. For default access modifier, Variables and methods can be declared without any modifiers.

Public Access Modifier - public
A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. However, if the public class we are trying to access is in a different package, then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses. This is why **main( )** has always been preceded by the **public** specifier.

Private Access Modifier - private
Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public getter methods are present in the class. Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

```
public class Logger {
private String format;
public String getFormat() {
return this.format;
}
public void setFormat(String  format) {
this.format =  format;
}
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly. So to make this variable available to the outside world, we defined two public methods: *getFormat*, which returns the value of format, and *setFormatString*, which sets its value.

Protected Access Modifier – protected
Protected applies only when inheritance is involved. Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. *The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.* Protected access gives the subclass a chance to use the helper method or variable, while preventing a non related class from trying to use it.

The following parent class uses protected access control, to allow its child class override ***openSpeaker*** method:

```
class AudioPlayer {
protected boolean openSpeaker(Speaker  sp) {
// implementation details
}
}
class StreamingAudioPlayer {
boolean openSpeaker(Speaker  sp) {
// implementation details
}
}
```

Here, if we define ***openSpeaker*** method as *private*, then it would not be accessible from any other class other than ***AudioPlayer***. If we define it as *public*, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, thats why we used *protected* modifier.
Usually, we want to restrict access to the data members of a class-allowing access only through methods. Also, there will be times when we want to define methods that 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 we 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( )**.

2.3.7. Understanding static

There will be times when we 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.

Both methods and variables can 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.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that 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, which prints a message and then initializes b 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.

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. 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 through their class name **StaticDemo**.

```
 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);
       }
 }
```

Output:

```
a = 42
b = 99
```

2.3.8. 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. 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 the 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.

2.3.9. 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 does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block. There are two types of nested classes: ***static*** and ***non-static***. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the 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.

Non Static Inner Class

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.
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);
}
} //end of inner class
} // end of outer class

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

Output:

```
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 an instance of **Inner** can be created 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**. (In general, an inner class instance must be created by an enclosing scope.) However, we can create an instance of **Inner** outside of **Outer** by qualifying its name with **Outer**, as in **Outer.Inner**.
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);
}
} //end of inner class
void showy() {
System.out.println(y); // error, y not known here!
}
}//end of outer class

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 inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope.

Static Nested class

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

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

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

```
public class Outer{
     static class Nested_Demo{
          public void my_method(){
               System .out.println("This is my nested class");
          }
     } //end of static inner class
     public static void main(String args[]){
          Outer.Nested_Demo nested=new Outer.Nested_Demo();
          nested.my_method();
     }
}//end of Outer class
Output:
This is my nested class
```

2.3.10. Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to **main( )**. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a **String** array passed to the **args** parameter of **main( )**. The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " + args[i]);
}
}
```

run:

```
java CommandLine this is a test 100 -1
```

Output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

REMEMBER All command-line arguments are passed as strings. We must convert numeric values to their internal forms manually. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String.
Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

2.4. Inheritance in Java

2.4.1. Introduction

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Inheritance is a form of *software reuse* in which a new class is created by absorbing an existing class's members and enhancing them with new or modified capabilities. With inheritance, we can save time during program development by basing new classes on existing proven and debugged high-quality software. This also increases the likelihood that a system will be implemented and maintained effectively.

When creating a class, rather than declaring completely new members, we can designate that the new class should inherit the members of an existing class. The existing class is called the **superclass**, and the new class is the **subclass**. Each subclass can become a *superclass* for future subclasses. A *subclass* can *add its own fields and methods*. Therefore, *a subclass is more specific than its superclass and represents a more specialized group of objects*. The subclass exhibits the behaviours of its superclass and can modify those behaviours so that they operate appropriately for the subclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements. This is why inheritance is sometimes referred to as ***specialization***.

The ***direct superclass*** is the superclass from which the subclass explicitly inherits. An indirect superclass is any class above the direct superclass in the class hierarchy, which defines the inheritance relationships between classes. In Java, the class hierarchy begins with class **Object** (in package *java.lang*), which every class in Java directly or indirectly extends (or "inherits from").

We distinguish between the *is-a relationship* and the *has-a relationship*. *Is-a* represents inheritance. In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass-e.g., *a car is a vehicle*. By contrast, *has-a* represents composition. In a *has-a* relationship, an object contains as members references to other objects-e.g., *a car has a steering wheel* (and a car object has a reference to a steering-wheel object). New classes can inherit from classes in class libraries. Organizations develop their own class libraries and can take advantage of others available worldwide. Some day, most new software likely will be constructed from *standardized reusable components*, just as automobiles and most computer hardware are constructed today. This will facilitate the development of more powerful, abundant and economical software.

2.4.2. Programming Basis

Inheritance provides the possibility to incorporate the definition of one class into another by using the **extends** keyword. The following program creates a superclass called **A** and a subclass called **B**.

```
class A {
    int i, j;
    void showij() {
        System.out.println("i and j: " + i + " " + j);
    }
    void setij(int m, int n)
    {
        i = m;
        j = n;
    }
}
// Create a subclass by extending class A.
class B extends A {
    int k;
    void showk() {
        System.out.println("k: " + k);
    }
    void setk(int p)
    {
        k = p;
    }
    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.setij(10,20);
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();
/* The subclass has access to all public members of its superclass.*/
        subOb.setij(7, 8);
        subOb.setk(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();
    }
}
```

```
Output:
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
```

Explanation: It is seen that, the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can call **setij()** and **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
}
```

Note: We can only specify one superclass for any subclass. Java does not support the inheritance of multiple super classes (***Multiple Inheritance***) into a single subclass. We can create a hierarchy of inheritance (**Hierarchical Inheritance and Multilevel Inheritance**) in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

2.4.3. 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**. 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.

```
// Create a superclass.
class ABC {
    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 XYZ extends ABC {
    int total;
    void sum() {
        total = i + j; // ERROR, j is not accessible here
    }
}
class Access {
    public static void main(String args[]) {
        XYZ subOb = new XYZ();
        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 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. We find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```
class Box {
    double width;
    double height;
    double depth;
   // 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 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, we 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. 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 Box does not define one.

2.4.4. Using super

When you 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. Each subclass constructor must implicitly or explicitly call its superclass constructor to initialize the instance variables inherited from the superclass. We do this by using the superclass constructor call syntax-keyword super, followed by a set of parentheses containing the superclass constructor arguments. **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.

2.4.4.1. Using super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of super:
`super(arg-list);`
Here, arg-list specifies any arguments needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass' constructor.

```
// 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 arguments **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.

2.4.4.2. 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();
}
}
```

Output:

```
i in superclass: 1
i in subclass: 2
```

2.4.5. Creating a Multilevel Hierarchy

We can build hierarchies that contain as many layers of inheritance. 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 super classes. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program.

```
class Car{
      public Car()
      {
            System.out.println("Class Car");
      }
      public void vehicleType()
      {
            System.out.println("Vehicle Type: Car");
      }
}
class Maruti extends Car{
      public Maruti()
      {
            System.out.println("Class Maruti");
      }
      public void brand()
      {
            System.out.println("Brand: Maruti");
      }
      public void speed()
      {
            System.out.println("Max Speed: 90Kmph");
      }
}
public class Maruti800 extends Maruti{

      public Maruti800()
      {
           System.out.println("Maruti Model: 800");
      }
      public void speed()
            {
               super.speed();
                 System.out.println("Max Speed: 80Kmph");
            }
      public static void main(String args[])
      {
            Maruti800 obj=new Maruti800();
            obj.vehicleType();
            obj.brand();
            obj.speed();
      }
}
```

Output:

```
Class Car
Class Maruti
Maruti Model: 800
Vehicle Type: Car
Brand: Maruti
Max Speed: 90Kmph
Max Speed: 80Kmph
```

2.4.6. 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:

```
// 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();
}
}
Output:
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor
```

2.4.7. Method Overriding

A derived class inherits methods that belong to the base class. However, if a derived class requires a different definition for an inherited method, the method may be redefined in the derived class. This is called **overriding** the method definition. When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. 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.

```
// 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() {
      super.show();  // this calls A's 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
}
}
```

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.

2.4.7.1. Rules for method overriding

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

2.4.7.2. Difference between overloading and overriding

SN	Method Overloading	Method Overriding
1	Method overloading is used *to increase the readability* of the program	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.

5	Overloading happens at *compile-time*. The binding of overloaded method call to its definition has happens at compile-time.	Overriding happens at *runtime*. Binding of overridden method call to its definition happens at runtime.
6	Static methods can be overloaded which means a class can have more than one static method of same name.	Static methods cannot be overridden, even if we declare a same static method in child class it has nothing to do with the same method of parent class.
7	Return type of method does not matter in case of method overloading, it can be same or different.	The overriding method can have more specific return type.

Table 2: Overloading vs Overriding

2.4.7.3. Dynamic Method Dispatch

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. 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 versions of the method are executed.

```
// 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
}
}

Output:
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 in turn 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**.

Note: Overridden methods in Java are similar to virtual functions in C++ and C# languages.

2.4.7.4. Why Overridden Methods?

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 superclasses 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.

Example

The following program creates a superclass called **Figure** that stores the dimensions of a two dimensional object. 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());
    }
}
```

```
Output:
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
Area for Figure is undefined.
Area is 0.0
```

2.5. Abstract Classes and Methods in Java

There are situations in which we want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we 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.

During the creation of our own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. We may have methods that must be overridden by the subclass in order for the subclass to have any meaning. We want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the ***abstract method***. It is requiring 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.

A class that is declared using "**abstract**" keyword is known as *abstract class*. It may or may not include *abstract methods* which means in abstract class we can have *concrete methods* (methods with body) as well along with *abstract methods* (without an implementation, without braces, and followed by a semicolon). *An abstract class can not be instantiated* (We are not allowed to create object of Abstract class).

2.5.1. Abstract class declaration

Specifying abstract keyword before the class during declaration, makes it abstract. Have a look at below code:

```
// Declaration using abstract keyword
abstract class AbstractDemo{
// Concrete method: body and braces
public void myMethod(){
//Statements here
}
// Abstract method: without body and braces
abstract public void anotherMethod();
}
```

Since abstract class allows concrete methods as well, it does not provide 100% abstraction. We can say that it provides partial abstraction. **Interfaces** are used for 100% abstraction.

Rules for Abstract Class

- If the class is having few ***abstract methods*** and few ***concrete methods***: declare it as abstract class.
- If the class is having only abstract methods: declare it as interface.
- Object creation of abstract class is not allowed
- We 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.

As discussed above, we cannot instantiate an abstract class. The following code throws an error.

```
abstract public class AbstractDemo{
public void myMethod(){
System.out.println("Hello");
}
abstract public void anotherMethod();
}
public class ConcreteDemo{
public void anotherMethod() {
System.out.print("Abstract method");
}
public static void main(String args[])
{
```

```
//Can't create object of abstract class - error!
AbstractDemo obj = new AbstractDemo();
obj.display();
}
}
```

Output:

Unresolved compilation problem: Cannot instantiate the type AbstractEx1

Note: The class that extends the *abstract class*, have to implement *all the abstract methods* of abstract class, else they can be declared abstract in the sub class as well.

Example (Abstract Method)

- Abstract method has no body.
- Always end the declaration with a semicolon();
- It must be overridden. An abstract class must be extended and in a same way abstract method must be overridden.
- Abstract method must be in a abstract class.

Note: The class which is extending abstract class must override (or implement) all the abstract methods.

```
// A Simple demonstration of abstract.
abstract class First {
    abstract void callme();
// concrete methods are still allowed in abstract classes
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}
class Second extends First {
    void callme() {
        System.out.println("B's implementation of callme.");
    }
}
class AbstractDemo {

    public static void main(String args[]) {
        Second s = new Second();
        s.callme();
        s.callmetoo();
    }
}
```

Output:

```
B's implementation of callme.
This is a concrete method.
```

Notice that no objects of class **First** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: ***class First*** 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

Key Points:

- An abstract class has no use until unless it is extended by some other class.
- If we declare an abstract method in a class then we must declare the class abstract as well.
- We can't have abstract method in a non-abstract class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
- Abstract class can have non-abstract method (concrete) as well.

2.5.2. Using final with Inheritance

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

2.5.2.1. Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when we 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 we 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.

2.5.2.3. Using final to Prevent Inheritance

Sometimes we 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. 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.

2.6. Interface in java

Interface looks like class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default. Java does not support multiple inheritance i.e. classes in java cannot have more than one superclass. For e.g. Definition of class like following in invalid in java

```
class A extends B extends C
{
……………..
……………..
}
```

A large number of real-life application requires the use of multiple inheritance where a class inherits properties and method from several distinct classes. For this purpose, java provides an alternate approach called ***interface*** to support the concept of *multiple inheritance*. A java *class cannot be a subclass of more than one superclass*, but it *can implement more than one interface thus supporting multiple inheritance*.

2.6.1. Defining Interface

An interface is basically a kind of class in the sense that interface contains methods and variables. But there is a major difference between a *class* and *interface*. The difference is that interfaces define only abstract methods and final fields. Hence interface has abstract method it does not specify any code to implement these methods. It is the responsibility of the class that implements an interface to define the code for implementation of these methods.The general form of interface definition is:

```
interface InterfaceName
{
Variable declaration;
Method declaration;
}
```

Here Interface is keyword and InterfaceName is any valid java name.
Variables are declared as:

```
static final return-type variableName = value;
```

Method are declared as:

```
return-type methodName (parameters);
```

Note: no any implementation code is provided in definition of interface; it is the responsibility of the class that implements the interface to provide the implementation code. For e.g.

```
interface Area
{
final static double pi = 3.14;
double computer (double x, double y);
void show();
}
```

2.6.2. Difference between class and interface

S.N.	Class	Interface
1	The members of class can have constant or variables.	The members of interface can have only constants.
2	Class can contain abstract as well as non-abstract method.	Interface can only have abstract method.

3	Class can contain the definition of method.	Interface cannot contain the definition of method. It is the responsibility of class that implements interface to provide the definition.
4	Class can be used to declare objects.	Interface cannot be used to declare objects
5	Class can use various access specifiers like public, private or protected	Interface can only use the public access specifier.

Table 3:Class vs Interface

2.6.3. Extending interface

Like classes interface can also be extended. The interface that extends another interface is called ***subinterface***. The new ***subinterface*** will inherit all the members of the ***superinterface***. Interface is inherited from another interface using keyword ***extends*** as shown below:

```
interface name2 extends name1
{
//body of name2
}
```

The situation where interface is used is illustrated by the following example. We can put all the constant in one interface and the methods in the other. This will enable us to use only the constants where the methods are not required and to use only methods where constants are not required. For example:

```
interface ItemConstants
{
static final int code = 1001;
static final String name =”fan”;
}
interface ItemMethods
{
void display();
}
```

Now if we want to use only constants in an interface called name1 then we can do:

```
Interface name1 extends ItemConstants
{
}
```

Now if we want to use only methods in an interface called name2 then we can do:

```
Interface name2 extends ItemMethods
{
}
```

Also if we want to use constants as well as method in an interface called name3 then we can do:

```
Interface name3 extends ItemConstants, ItemMethods
{
}
```

Note: When interface extends another *interface*, *subinterface* connot define the methods declared in *superinterface*. It is the responsibility of any class that implements the derived interface to define all the methods.

2.6.4. Implementing interfaces

Interfaces are normally used as "*superclasses*" whose properties are inherited by classes. Class can implement interface using keyword **implements** as shown below:

```
class ClassName implements InterfaceName
{
//body of class
}
```

A class can extend another class and at the same time implement an interface. This is done as:

```
class ClassName extends SuperClass implements InterfaceName
{
//body of class
}
```

To implement multiple interface, we implement class as:

```
class ClassName implements InterfaceName1, InterfaceName2, ………
{
//body of class
}
```

Example program:

```
interface Area {
    final static double pi = 3.14;
    double compute(double x, double y);
}
class Rectangle implements Area {
    public double compute(double x, double y) {
        return (x * y);
    }
}
class Circle implements Area {
    public double compute(double x, double y) {
        return (pi * x * x);
    }
}
class InterfaceTest {
    public static void main(String args[]) {
        Rectangle rect = new Rectangle();
        Circle circ = new Circle();
        Area area;//interface object
        area = rect;//area refers to rect object
        System.out.println("Area of rectangle="+area.compute(10, 45));
        area = circ;//area refers to circ object
        System.out.println("Area of rectangle="+area.compute(10, 0));
    }
}
```

```
Output:
Area of rectangle =450.0
Area of rectangle =314.0
```

Key points: Here are the key points to remember about interfaces:

- We can't instantiate an interface in java.
- Interface provides complete abstraction as none of its methods can have body. On the other hand, abstract class provides partial abstraction as it can have abstract and concrete (methods with body) methods both.
- implements keyword is used by classes to implement an interface.
- While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- Class implementing any interface must implement all the methods, otherwise the class should be declared as "abstract".
- Interface cannot be declared as private or protected.
- All the interface methods are by default abstract and public.
- Variables declared in interface are public, static and final by default.

2.7. Packages

One way of reusing the code already created is by extending the classes and implementing the interfaces. But this is limited to reusing the classes within a program. What if we want to use classes from other programs without physically copying them into the program under development. This can be achieved in java by using the concept of ***packages***. Package is similar to header file and class libraries in other language.
The package is both a naming and a visibility control mechanism. We can define classes inside a package that are not accessible by code outside that package. We can also define class members that are only exposed to other members of the same package. This allows our classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world. Packages are used in Java in order to *prevent naming conflicts*, to *control access*, to *make searching/locating and usage of classes*, *interfaces*, *enumerations* and *annotations* easier, etc.
A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Packages in java are classified into two categories:
1. Java API packages
2. User defined packages

1. Java API packages

Java API packages provided a large number of classes grouped into different packages according to functionality. Most of the time we use classes from Java API packages. Figure below shows the java API packages that are frequently used in the programs:

Package	Description
java.applet	The Java Applet Package contains a class and several interfaces required to create Java applets-programs that execute in web browsers.
java.awt	The Java *Abstract Window Toolkit* Package contains the classes and interfaces required to create and manipulate GUIs in early versions of Java. In current versions, the Swing GUI components of the javax.swing packages are typically used instead.
java.awt.event	The Java Abstract Window Toolkit Event Package contains classes and interfaces that enable event handling for GUI components in both the java.awt and javax.swing packages.
java.awt.geom	The Java 2D Shapes Package contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities.
java.io	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data.
java.lang	The Java Language Package contains classes and interfaces that are required by many Java programs. This package is imported by the compiler into all programs.
java.net	The Java Networking Package contains classes and interfaces that enable programs to communicate via computer networks like the Internet.
java.sql	The JDBC Package contains classes and interfaces for working with databases.

java.util	The Java Utilities Package contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class Random) and the storing and processing of large amounts of data.
java.util. concurrent	The Java Concurrency Package contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel.
javax.media	The Java Media Framework Package contains classes and interfaces for working with Java's multimedia capabilities. (
javax.swing	The Java Swing GUI Components Package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs.
projavax. swing.event	The Java Swing Event Package contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package javax.swing.
javax.xml.ws	The JAX-WS Package contains classes and interfaces for working with web services in Java.

Table 4: Java API Packages

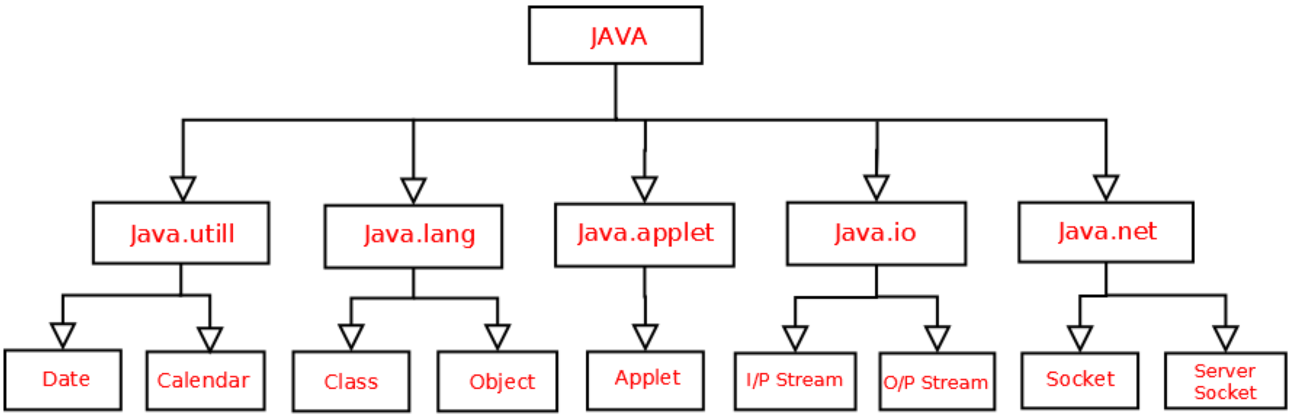


Figure 3: Java Packages Hierarchy

The packages are organized in a hierarchical structure. Since the package is collection of either related classes and/or interface the packages contain class. For e.g. the package *awt* contains various classes required for implementing graphical user interface.

Accessing class from packages

There are two ways of accessing the classes stored in a package.

3. If we want to use only one class from a package we specify the fully classified class name of the class that we want to use. This is done by using the package name containing the class and then appending it with dot operator. For e.g. to use class called *Color* in *awt* package we write: *import java.awt.Colour;*
4. If we want to use many classe from a package we use asterisk(*). Asterisk represents all the class in a file. So if we want to use many classes from package from package awt. The we write: *import java.awt.*;*

Import statement must be the first statement after the package name.

2.7.1. Creating user defined packages

We can create our own package in java. To create package in java we must first declare the name of the package using the package keyword followed by package name and then we write the definition of the classes that we want to be in that package. For example, if we want class named *FirstClass* to be inside package *firstPackage* then we write:

```
package firstPackage;
```

```
public class FirstClass
{
//body of class
}
```

Note: We can also create package hierarchy. This is done by specifying multiple names in a package statement, separated by dots.

```
E.g.  package firstPackage.secondPackage;
```

This approach allows us to group related classes into a package and then group related package into larger package. In any package many class can be included but there should be only one class with public modifier. A package hierarchy must be reflected in the file system of our Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in *java\awt\image* in a Windows environment. We cannot rename a package without renaming the directory in which the classes are stored.

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting *.class* file is also in the *MyPack* directory. Then, try executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance
```

We need to be in the directory above MyPack when we execute this command. As explained, *AccountBalance* is now part of the package MyPack. This means that it cannot be executed by itself. That is, we cannot use this command line: java AccountBalance
AccountBalance must be qualified with its package name.

2.7.2. The import Keyword

If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.

Example
Here, a class named **Boss** is added to the *payroll* package that already contains **Employee**. The **Boss** can then refer to the **Employee** class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;
public class Boss
{
public void payEmployee(Employee e)
{
e.mailCheck();
}
}
```

What happens if the **Employee** class is not in the *payroll* package? The **Boss** class must then use one of the following techniques for referring to a class in a different package.
The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

2.7.3. Access Protection

We know that access to a private member of a class is granted only to other members of that class. Packages add another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. While Java's access control mechanism may seem complicated, we can simplify it as follows.

- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
- If we want to allow an element to be seen outside our current package, but only to classes that subclass our class directly, then declare that element protected.

Below table applies only to members of classes. A non-nested class has only two possible access levels: **default** and **public**. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 5: Class Member Access

Example

```
package MyPack;
/* Now, the Balance class, its constructor, and its show() method are
public. This means that they can be used by non-subclass code outside
their package.
*/
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n;
bal = b;
}
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```

The Balance class is now **public**. Also, its constructor and its show( ) method are public, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the Balance class:

```
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
/* Because Balance is public, we may use Balance class and call its
constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

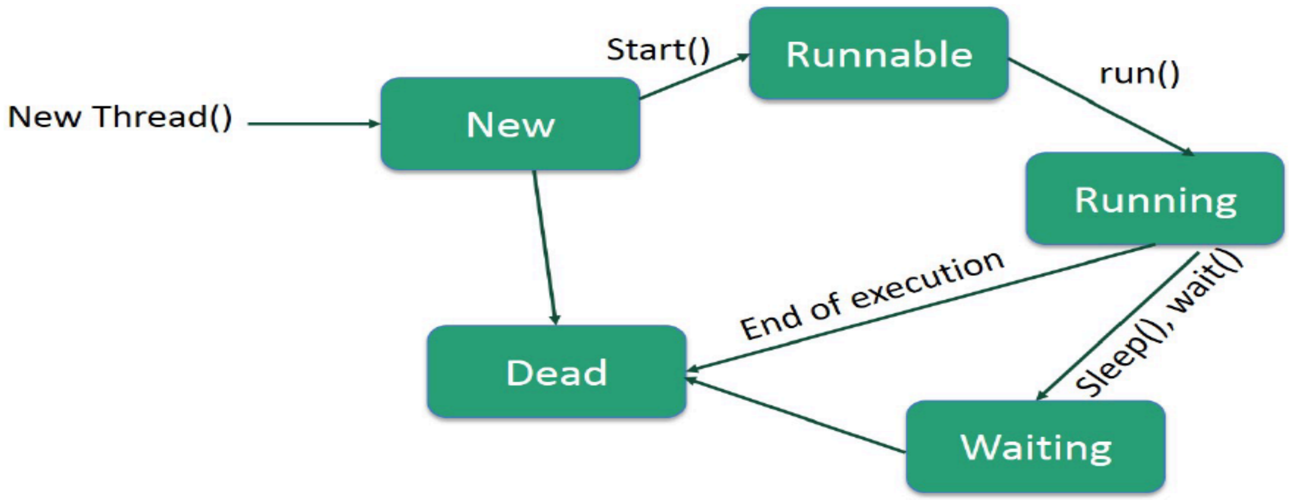
2.8. Multithreading in Java

Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when our computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where we can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application. Multi-threading enable us to write in a way where multiple activities can proceed concurrently in the same program.

2.8.1. 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. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

- **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.

2.8.2. Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java thread priorities are in the range between ***MIN_PRIORITY(***a constant of 1) and **MAX_PRIORITY**(a constant of 10) . By default, every thread is given priority **NORM_PRIORITY** (a constant of 5).
Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

2.8.2.1. Creating a Thread by Implementing Runnable Interface

If a class is intended to be executed as a thread then it can achieve this by implementing **Runnable** interface.
Step 1: As a first step we need to implement a **run()** method provided by **Runnable** interface. This method provides entry point for the thread and we will put our complete business logic inside this method.
Following is simple syntax of run method:
public void run( )
Step 2: At second step we will instantiate a Thread object using the following constructor:
Thread(Runnable threadObj, String threadNam e);
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 Thread object is created, we can start it by calling **start**() method, which executes a call to
run method. Following is simple syntax of start method:
void start( );

Example Program

```
class RunnableDemo implements Runnable {

    private Thread t;
    private String threadName;

    RunnableDemo(String name) {
        threadName = name;
        System.out.println("Creating " + threadName);
    }

    public void run() {
        System.out.println("Running " + threadName);
        try {
            for (int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + "
interrupted.");
        }
        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 ThreadDemo {

    public static void main(String args[]) {
        RunnableDemo R1 = new RunnableDemo("Thread-1");
        R1.start();
        RunnableDemo R2 = new RunnableDemo("Thread-2");
        R2.start();
    }
}
```

Output

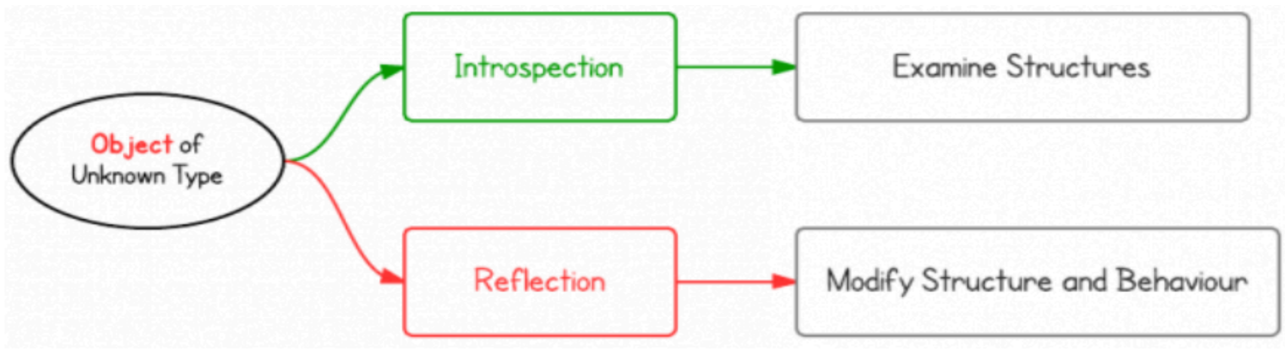
```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-2, 1
Thread: Thread-1, 1
Thread Thread-2 exiting.
Thread Thread-1 exiting.
```

2.9. Reflection in Java

Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.
Java Reflection is quite powerful and can be very useful. For instance, when mapping objects to tables in a database at runtime. Or, when mapping the statements in a script language to method calls on real objects at runtime.
This concept is often mixed with introspection. The following are their definitions:

- Introspection is the ability of a program to examine the type or properties of an object at runtime.
- Reflection is the ability of a program to examine and modify the structure and behavior of an object at runtime.

From their definitions, introspection is a subset of reflection. Some languages support introspection, but do not support reflection, e.g., C++.



Introspection Example: The **instanceof** operator determines whether an object belongs to a particular class.

```
if(obj instanceof Dog){
Dog d = (Dog)obj;
d.bark();
}
```

Reflection Example: The **Class.forName()** method returns the Class object associated with the class/interface with the given name(a string and full qualified name). The **forName** method causes the class with the name to be initialized.

```
// with reflection
Class<?> c = Class.forName("classpath.and.classname");
Object dog = c.newInstance();
Method m = c.getDeclaredMethod("bark", new Class<?>[0]);
m.invoke(dog);
```

In Java, reflection is more about introspection, because we cannot change structure of an object. There are some APIs to change accessibilities of methods and fields, but not structures.

Why do we need reflection?

Reflection enables us to:

1. Examine an object's class at runtime.
2. Construct an object for a class at runtime.
3. Examine a class's field and method at runtime.
4. Invoke any method of an object at runtime.
5. Change accessibility flag of Constructor, Method and Field

Example 1: Get class name from object

```
import java.lang.reflect.*;
public class ReflectionHelloWorld {

    public static void main(String[] args) {
        Hello f = new Hello();
        System.out.println(f.getClass().getName());
    }
}
class Hello {
    public void print() {
        System.out.println("abc");
    }
}
```

Output:

```
Reflection.Hello
```

Example 2: Invoke method on unknown object
For the code example below, image the types of an object is unknown. By using reflection, the code can use the object and find out if the object has a method called "**print**" and then call it.

```
import java.lang.reflect.Method;
public class ReflectionDemo {
    public static void main(String[] args) {
        Foo f = new Foo();
        Method method;
        try {
            method = f.getClass().getMethod("print", new Class<?>[0]);
            method.invoke(f);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
class Foo {

    public void print() {
        System.out.println("abc");
    }
}
```

Output:

```
abc
```

Chapter 3: Exception, Stream and I/O

3.1. Exception

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled. An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually-typically through the use of error codes, and so on. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world. An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

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

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. Based on these, we have three categories of Exceptions.

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

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

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

3.1.1. Error and Exception

Both Error and Exception are derived from *java.lang.Throwable* in Java but main difference between Error and Exception is kind of error they represent. *java.lang.Error* represent errors which are generally can not be handled and usually refer catastrophic failure e.g. *running out of System resources*, some examples of Error in Java are *java.lang.OutOfMemoryError* or *Java.lang.NoClassDefFoundError* and *java.lang.UnSupportedClassVersionError*. On the other hand *java.lang.Exception* represent errors which can be catch and dealt e.g. IOException which comes while performing I/O operations i.e. reading files and directories.

3.1.2. Exception-Handling Fundamentals

A Java *exception* is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an *object* representing that exception is created and thrown in the method that caused the error. That method may choose to handle the

exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the *Java run-time* system, or they can be manually generated by our code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
Java exception handling is managed via five keywords: ***try, catch, throw, throws, and finally***. Briefly, here is how they work.

- Program statements that we want to monitor for exceptions are contained within a try block.
- If an exception occurs within the try block, it is thrown.
- The code can catch this exception (using catch) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword *throw*.
- Any exception that is thrown out of a method must be specified as such by a *throws* clause.
- Any code that absolutely must be executed after a try block completes is put in a *finally* block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
finally {
// block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

3.1.3. Exception Types

All exception types are subclasses of the built-in class ***Throwable***. Thus, *Throwable* is at the top of the exception class hierarchy. Immediately below *Throwable* are two subclasses that partition exceptions into two distinct branches. One branch is headed by ***Exception***. This class is used for exceptional conditions that user programs should catch. This is also the class that we will subclass to create our own custom exception types. There is an important subclass of Exception, called *RuntimeException*. Exceptions of this type are automatically defined for the programs that we write and include things such as division by zero and invalid array indexing.

The other branch is topped by ***Error***, which defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type *Error are used by the Java run-time* system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

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

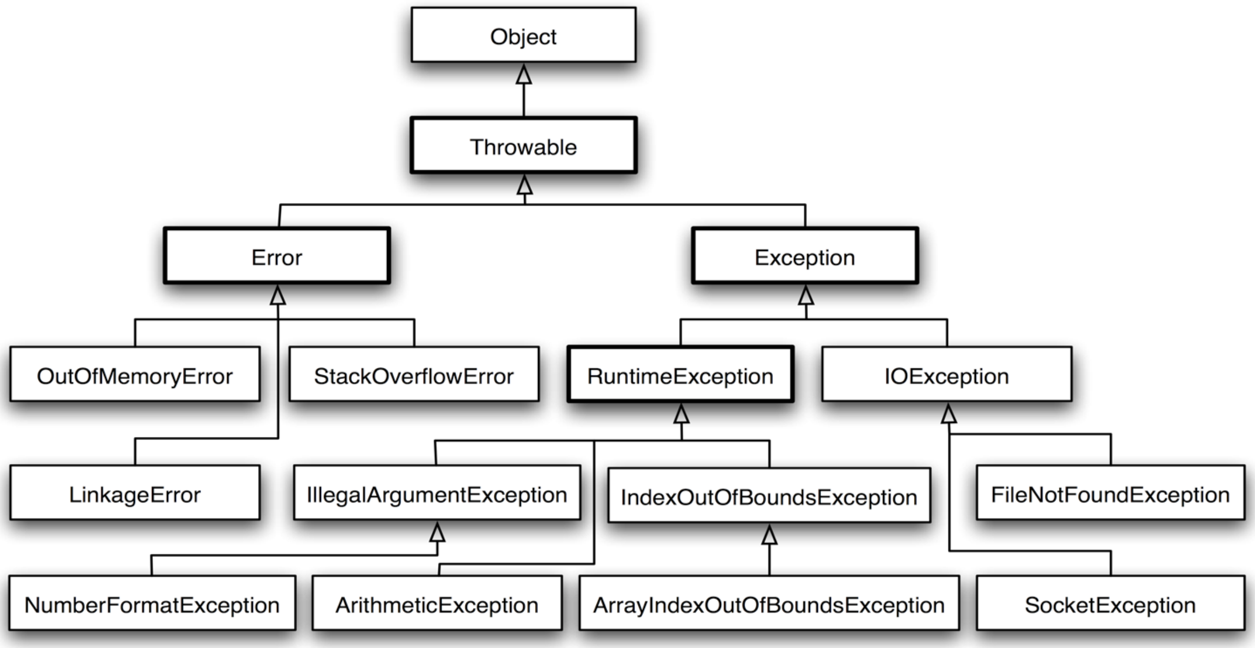


Figure 4: Java Exception Hierarchy

3.1.4. Built-in Exceptions

Java defines several exception classes inside the standard package java.lang.The most general of these exceptions are subclasses of the standard type *RuntimeException*. Since *java.lang* is implicitly imported into all Java programs, most exceptions derived from *RuntimeException* are automatically available.Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked *RuntimeException*.

SN	Exception	Description
1	ArithmeticException	Arithmetic error, such as divide-by-zero.
2	ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
3	ArrayStoreException	Assignment to an array element of an incompatible type.
4	ClassCastException	Invalid cast.
5	IllegalArgumentException	Illegal argument used to invoke a method.
6	IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
7	IllegalStateException	Environment or application is in incorrect state.
8	IllegalThreadStateException	Requested operation not compatible with the current thread state.
9	IndexOutOfBoundsException	Some type of index is out-of-bounds.
10	NegativeArraySizeException	Array created with a negative size.
11	NullPointerException	Invalid use of a null reference.
12	NumberFormatException	Invalid conversion of a string to a numeric format.
13	SecurityException	Attempt to violate security.
14	StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
15	UnsupportedOperationException	An unsupported operation was encountered.

Table 6: Java's Unchecked RuntimeException Subclasses Defined in java.lang

Following is the list of Java Checked Exceptions Defined in *java.lang*.

SN	Exception	Description
1	ClassNotFoundException	Class not found.
2	CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
3	IllegalAccessException	Access to a class is denied.
4	InstantiationException	Attempt to create an object of an abstract class or interface.
5	InterruptedException	One thread has been interrupted by another thread.
6	NoSuchFieldException	A requested field does not exist.
7	NoSuchMethodException	A requested method does not exist.

Table 7: Java's Checked Exceptions Defined in java.lang

3.1.5. Uncaught Exceptions

It is useful to see what happens when we don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to *divide by zero*, it constructs a new exception object and then throws this exception. This causes the execution of *Exc0* to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the default handler provided by the Java run-time system catches the exception. The default handler will ultimately process any exception that is not caught by our program. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

3.1.6. Using *try* and *catch*

Although the default exception handler provided by the *Java run-time system* is useful for debugging, we usually want to handle an exception our self. Doing so provides two benefits. First, *it allows us to fix the error*. Second, *it prevents the program from automatically terminating*. To guard against and handle a run-time error, simply enclose the code that we want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that we wish to catch. Following block of code shows that the *ArithmeticException* generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
} //End of Catch
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

3.1.7. Multiple *catch* Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, we can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

```
class MultiCatch {
      public static void main(String args[]) {
      try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
            } catch(ArithmeticException e) {
                  System.out.println("Divide by 0: " + e);
            } catch(ArrayIndexOutOfBoundsException e) {
                  System.out.println("Array index oob: " + e);
            }
      System.out.println("After try/catch blocks.");
      }
}
```

This program will cause a division-by-zero exception if it is started with no commandline arguments, since a will equal zero. It will survive the division if we provide a command-line argument, setting **a** to something larger than zero. But it will cause an *ArrayIndexOutOfBoundsException*, since the int array c has a length of 1, yet the program attempts to assign a value to c[42].
Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

When using multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. *Further, in Java, unreachable code is an error*. For example, consider the following program:

```
class SuperSubCatch {
      public static void main(String args[]) {
            try {
                  int a = 0;
                  int b = 42 / a;
            }catch(Exception e) {
                  System.out.println("Generic Exception catch.");
            }
            /* This catch is never reached because ArithmeticException is
      a subclass of Exception. */
            catch(ArithmeticException e) { // ERROR - unreachable
                  System.out.println("This is never reached.");
            }
      }
}
```

If we try to compile this program, we will receive an error message stating that the *second catch statement is unreachable because the exception has already been caught*. Since *ArithmeticException* is a subclass of Exception, the first catch statement will handle all Exception-based errors, including *ArithmeticException*. This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

3.1.8. Nested *try* Statements

The *try* statement can be nested. That is, a *try* statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested try statements:

```
// An example of nested try statements.
class NestTry {
     public static void main(String args[]) {
          try {
                int a = args.length;
/* If no command-line args are present,the following statement will
generate a divide-by-zero exception. */
                int b = 42 / a;
                System.out.println("a = " + a);
                try { // nested try block
/* If one command-line arg is used,then a divide-by-zero exception will be
generated by the following code. */
                     if(a==1)
                          a = a/(a-a); // division by zero
/* If two command-line args are used,then generate an out-of-bounds
exception. */
                     if(a==2) {
                          int c[] = { 1 };
                          c[42]=99;//generate an out-of-bounds exception
                     }
                } catch(ArrayIndexOutOfBoundsException e) {
                    System.out.println("Array index out-of-bounds:"+ e);
                }
                } catch(ArithmeticException e) {
                     System.out.println("Divide by 0: " + e);
          }
     }
}
```

The program works as follows. When we execute the program with no command-line arguments, a *divide-by-zero exception* is generated by the outer try block. Execution of the program with one command-line argument generates a ***divide-by-zero*** exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled. If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

3.1.9. throw

However, it is possible for our program to throw an exception explicitly, using the ***throw*** statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, *ThrowableInstance* must be an object of type *Throwable* or a subclass of *Throwable*. Primitive types, such as *int* or *char*, as well as *non-Throwable* classes, such as String and Object, cannot be used as exceptions. There are two ways that we can obtain a *Throwable* object: using a parameter in a *catch* clause, or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

```
// Demonstrate throw.
class ThrowDemo {
      static void demoproc() {
            try {
                  throw new NullPointerException("demo");
                  } catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc.");
                        throw e; // rethrow the exception
                  }
            }
      public static void main(String args[]) {
      try {
            demoproc();
            } catch(NullPointerException e) {
                  System.out.println("Recaught: "+e);
            }
      }
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another *exceptionhandling* context and immediately throws a new instance of *NullPointerException*, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

3.1.10. throws

If a method does not handle a checked exception, the method must declare it using the *throws* keyword. The throws keyword appears at the end of a method's signature. If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. We do this by including a *throws* clause in the method's declaration. A *throws* clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type *Error or RuntimeException*, or any of their subclasses. All other exceptions that a method can throw must be declared in the *throws* clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

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

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
} //End of catch
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

3.1.11. finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.

The *finally* block follows a *try* block or a *catch* block. A *finally* block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows us to run any cleanup type statements that we want to execute, no matter what happens in the protected code. The *finally* block will execute whether or not an exception is thrown. If an exception is thrown, the *finally* block will execute even if no *catch* statement matches the exception. Any time a method is about to return to the caller from inside a *try/catch* block, via an uncaught exception or an explicit return statement, the *finally* clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The *finally* clause is optional. However, each try statement requires at least one catch or a finally clause.
Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

```
// Demonstrate finally.
class FinallyDemo {
      // Through an exception out of the method.
      static void procA() {
            try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
            } finally {
            System.out.println("procA's finally");
            }
            }
            // Return from within a try block.
            static void procB() {
                  try {
                  System.out.println("inside procB");
                  return;
                  } finally {
                  System.out.println("procB's finally");
                  }
            }
            // Execute a try block normally.
            static void procC() {
                  try {
                  System.out.println("inside procC");
                  } finally {
                  }
            }
            public static void main(String args[]) {
            try {
                  procA();
                  } catch (Exception e) {
                  System.out.println("Exception caught");
            }
            procB();
            procC();
      }
}
```

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The *finally* clause is executed on the way out. procB( )'s try statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed.
Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

3.1.12. User-defined Exceptions

We can create our own exceptions in Java. Following is the key points to understand before we write our own exception classes:

- All exceptions must be a child of *Throwable*.
- If we want to write a checked exception that is automatically enforced by the Handle or Declare Rule, we need to extend the Exception class.
- If we want to write a *runtime exception*, we need to extend the *RuntimeException* class.

Syntax

```
class MyException extends Exception{
      //class definition
}
```

The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString() method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}

class ExceptionDemo {
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays the value of the exception. The ExceptionDemo class defines a method named compute( ) that throws a MyException object. The exception is thrown when compute( )'s integer parameter is greater than 10. The main( ) method sets up an exception handler for MyException, then calls compute( ) with a legal value (less than 10) and an illegal one to show both paths through the code.

Output:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

3.1.13. Debugging

3.1.13.1. Loop Bug

The two most common kinds of loop errors are unintended *infinite loops* and *off-by-one errors*.

Infinite Loop: A while loop, do-while loop, or for loop does not terminate as long as the controlling Boolean expression evaluates to true . This Boolean expression normally contains a variable that will be changed by the loop body, and usually the value of this variable eventually is changed in a way that makes the Boolean expression false and therefore terminates the loop. However, if we make a mistake and write our program so that the Boolean expression is always true, then the loop will run forever. A loop that runs forever is called an infinite loop.

off-by-one error: The loop repeats the loop body *one too many or one too few times*. These sorts of errors can result from carelessness in designing a controlling Boolean expression. For example, if we use less-than when we should use less-than-or-equal, this can easily make our loop iterate the body the wrong number of times.

Use of == to test for equality in the controlling Boolean expression of a loop can often lead to an off-by-one error or an infinite loop. This sort of equality testing can work satisfactorily for integers and characters, but is not reliable for floating-point numbers. This is because the floating-point numbers are approximate quantities, and == tests for exact equality. The result of such a test is unpredictable. When comparing floating-point numbers, always use something involving less-than or greater-than, such as <= ; do not use == or != . Using == or != to test floating-point numbers can produce an off-by-one
error or an unintended infinite loop or even some other type of error. Even when using integer variables, it is best to avoid using == and != and to instead use something involving less-than or greater-than.

3.1.13.2. Tracing Variables

One good way to discover errors in a loop or any kind of code is to trace some key variables. Tracing variables means watching the variables change value while the program is running. Most programs do not output each variable's value every time the variable changes, but being able to see all of these variable changes can help us to debug your program.

3.1.13.3. General Debugging Techniques

Tracing errors can sometimes be a difficult and time-consuming task. It is not uncommon to spend more time debugging a piece of code than it took to write the code in the first place. If we are having difficulties finding the source of our errors, then there are some general debugging techniques to consider. If the program is giving incorrect output values, then we should examine the source code, different test cases using a range of input and output values, and the logic behind the algorithm itself.

Determining the precise cause and location of a bug is one of the first steps in fixing the error. Examine the input and output behavior for different test cases to try to localize the error. A related technique is to trace variables to show what code the program is executing and what values are contained in key variables. We might also focus on code that has recently changed or code that has had errors before.

Finally, we can also try removing code. If we comment out blocks of code and the error still remains, then the culprit is in the uncommented code. The process can be repeated until the location of the error can be pinpointed. The /* and */ notation is particularly useful to comment out large blocks of code. After the error has been fixed, it is easy to remove the comments and reactivate the code.

The first mistakes we should look for are common errors that are easy to make. Examples of common errors include off-by-one errors, comparing floating-point types with == , adding extra semicolons that terminate a loop early, or using == to compare strings for equality.

3.2. I/O Streams

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays. Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways. No matter how they work internally, all streams present the same simple model to programs that use them: *A stream is a sequence of data*. A program uses an *input stream* to read data from a source, one item at a time:

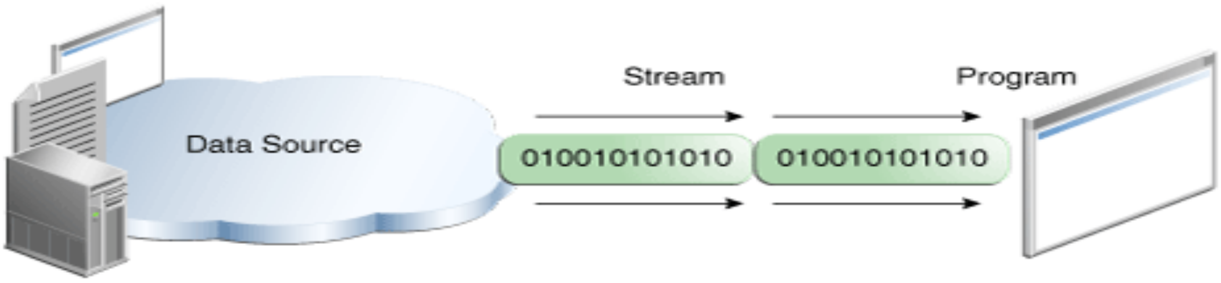


Fig: Reading information into a program

A program uses an *output stream* to write data to a destination, one item at time:

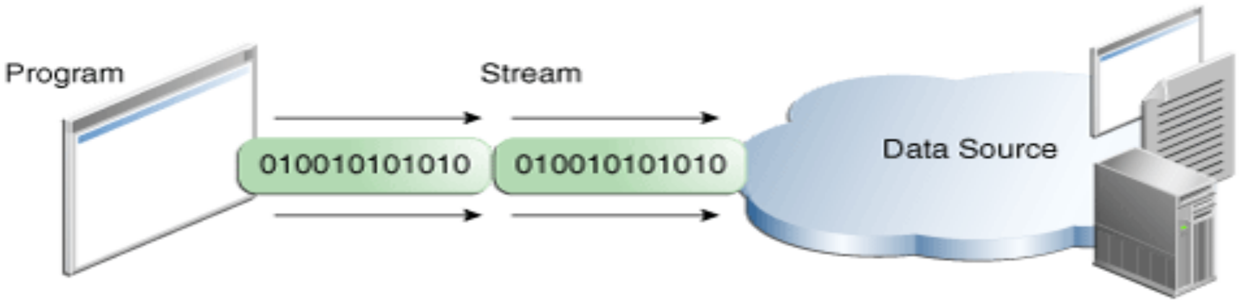


Fig: Writing information from a program.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array. The following figure illustrates the hierarchy of classes to deal with Input and Output streams.

3.2.1. 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:

```
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();
}
} //End of finally
} //End of main
} //End of class
```

3.2.2. 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.

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();
}
} //End of finally
} //End of main
} //End of class
```

3.2.3. 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. 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":

```
import java.io.*;
public class ReadConsole {
public static void main(String args[]) throws IOException
{
InputStreamReader cin = null;
try {
cin = new InputStreamReader(System.in);
System.out.println("Enter characters, 'q' to quit.");
char c;
do {
c = (char) cin.read();
System.out.print(c);
} while(c != 'q');
}finally {
if (cin != null) {
cin.close();
}
} //End of finally
} //End of main
} //End of class
```

3.2.4. Reading and Writing Files

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.

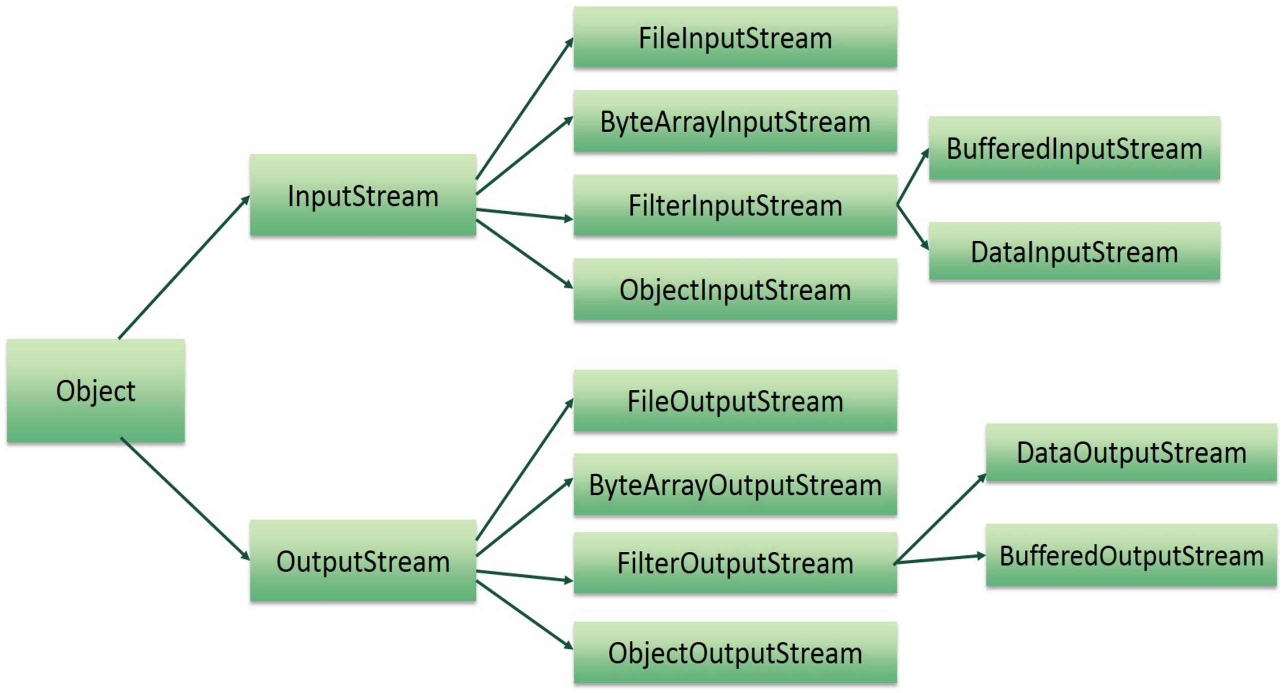


Figure 5:Classes to deal with Input and Output streams

3.2.4.1. 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 we 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.

SN	Methods with 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 end of 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 end of 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.

Table 8: InputStream Helper Methods

3.2.4.2. 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. 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 we 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.

SN	Methods with 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*.

Table 9: Helper Methods of OutputStream

Following is the example to demonstrate InputStream and OutputStream.

```
import  java. io.* ;
public class  FileStream Test{
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");
} //End of catch
} //End of main
} //End of Class
```

3.2.5. Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Using a fixed-length record format is a good choice if you need to store data of the same type. However, objects that you create in an object-oriented program are rarely all of the same type. For example, you may have an array called *staff* that is nominally an array of *Employee* records but contains objects that are actually instances of a child class such as *Manager*. If we want to save files that contain this kind of information, we must first save the type of each object and then the data that defines the current state of the object. When we read this information back from a file, we must:

- Read the object type;
- Create a blank object of that type;
- Fill it with the data that we stored in the file.

It is entirely possible (if very tedious) to do this by hand. Sun Microsystems developed a powerful mechanism that allows this to be done with much less effort. This mechanism is called object serialization, almost completely automates what was previously a very tedious process.

3.2.6. File 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. The File object represents the actual file/directory on the disk. Following is the list of constructors to create a File object.

SN	Methods with 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.

SN.	Methods with 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 non existent 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.

Following is an example to demonstrate File object.

```
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();
}
}
}
```

Chapter 4: Applets and Application

4.1. Applet Fundamentals

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity. Applets differ from console-based applications in several key areas.

- 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.

Example

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
      public void paint(Graphics g) {
      g.drawString("A Simple Applet", 20, 20);
      }
}
```

1. This applet begins with two ***import*** statements. The first imports the *Abstract Window Toolkit* (AWT) classes. Applets interact with the user (either directly or indirectly) through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical user interface.
2. The second ***import*** statement imports the ***applet*** package, which contains the class ***Applet***. Every applet that we create must be a subclass of ***Applet***.
3. The next line in the program declares the class ***SimpleApplet***. This class must be declared as ***public***, because it will be accessed by code that is outside the program.
4. Inside ***SimpleApplet***, ***paint( )*** is declared. This method is defined by the AWT and must be overridden by the *applet*. paint( ) is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, *the window in which the applet is running can be overwritten by another window and then uncovered*. Or, *the applet window can be minimized and then restored*. ***paint( )*** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
5. Inside paint( ) is a call to ***drawString( ),*** which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

void drawString(String message, int x, int y)

6. Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to ***drawString( )*** in the applet causes the message “A Simple Applet” to be displayed beginning at location 20,20.
7. Notice that the applet ***does not have a main( )*** method. Unlike Java programs, applets do not begin execution at main( ). In fact, most applets don’t even have a main( ) method. Instead, an applet begins execution ***when the name of its class is passed to an applet viewer*** or to a network browser.
8. After we enter the source code for ***SimpleApplet***, compile in the same way that we have been compiling programs. However, running ***SimpleApplet*** involves a different process. In fact, there are two ways in which you can run an applet:
   - Executing the applet within a Java-compatible web browser.
   - Using an applet viewer, such as the standard tool, ***appletviewer***. An applet viewer executes our applet in a window. This is generally the fastest and easiest way to test our applet.

To execute an applet in a web browser, we need to write a short HTML text file that contains a tag that loads the applet. Currently, Sun recommends using the APPLET tag for this purpose. Here is the HTML file that executes ***SimpleApplet***:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The width and height statements specify the dimensions of the display area used by the applet. After the creation of this file, we can execute our browser and then load this file, which causes ***SimpleApplet*** to be executed.

To execute ***SimpleApplet*** with an applet viewer, we may also execute the HTML file shown earlier. For example, if the preceding HTML file is called ***RunApp.html***, then the following command line will run SimpleApplet:

```
C:\>appletviewer RunApp.html
```

However, a more convenient method exists that we can use to speed up testing. Simply include a comment at the head of Java source code file that contains the APPLET tag. By doing so, our code is documented with a prototype of the necessary HTML statements, and we can test our compiled applet merely by starting the applet viewer with our Java source code file. The ***SimpleApplet*** source file looks like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
      public void paint(Graphics g) {
      g.drawString("A Simple Applet", 20, 20);
      }
}
```

Key Points

1. Applets do not need a main( ) method.
2. Applets must be run under an applet viewer or a Java-compatible browser.
3. User I/O is not accomplished with Java’s stream I/O classes. Instead, applets use the interface provided by the AWT or Swing.
4. All applets are subclasses (either directly or indirectly) of Applet.
5. Applets are not stand-alone programs. Instead, they run within either a web browser or an

applet viewer.

6. Output is handled with various AWT methods, such as ***drawString()***,which outputs a string to a specified X,Y location, not by ***system.out.println()***.
7. A Java-enabled web browser will execute the applet when it encounters the APPLET tag within the HTML file.

4.2. Two Types of Applet

There are two varieties of applets.

1. The first type of applets uses the ***Abstract Window Toolkit (AWT)*** to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created.
2. The second type of applets is those based on the Swing class ***JApplet***. Swing applets use the Swing classes to provide the GUI. *Swing offers a richer and often easier-to-use user interface than does the AWT*. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required. Thus, both AWT- and Swing-based applets are valid. Because ***JApplet*** inherits ***Applet***, all the features of ***Applet*** are also available in ***JApplet***.

4.3. The Applet Class

The ***Applet*** class defines the methods shown in following table. ***Applet*** provides all necessary support for applet execution, such as *starting and stopping*. It also provides methods that *load and display images*, and *methods that load and play audio clips*. ***Applet*** extends the ***AWT class Panel***. In turn, ***Panel*** extends ***Container***, which extends ***Component***. These classes provide support for Java's window-based, graphical interface. Thus, ***Applet*** provides all of the necessary support for window-based activities.

Methods	**Description**
void destroy( )	Called by the browser just before an applet is terminated.
AccessibleContext getAccessibleContext( )	Returns the accessibility context for the invoking object.
AppletContext getAppletContext( )	Returns the context associated with the applet.
String getAppletInfo( )	Returns a string that describes the applet.
AudioClip getAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
Image getImage(URL url)	Returns an Image object that encapsulates the image found at the location specified by url.
Locale getLocale( )	Returns a Locale object that is used by various locale sensitive classes and methods.
String getParameter(String paramName)	Returns the parameter associated with ***paramName.null*** is returned if the specified parameter is not found.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns true if the applet has been started. It returns false if the applet has been stopped.
void play(URL url)	If an audio clip is found at the location specified by url, the clip is

	played.
void play(URL url, String clipName)	If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after init( ) when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls start( ).
void showStatus(String str)	Displays str in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.

Table 10:The Methods Defined by Applet

2.8. Applet Architecture

An applet is a window-based program. As such, its architecture is different from the console-based programs. First, applets are event driven. It is important to understand in a general way that the event-driven architecture impacts the design of an applet.
An applet resembles a set of *interrupt service routines*. Here is how the process works. An applet waits until an ***event*** occurs. The ***run-time system*** notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point. For the most part, applet should not enter a "mode" of operation in which it maintains control for an extended period. Instead, *it must perform specific actions in response to events and then return control to the run-time system*. In those situations in which applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), we must start an additional thread of execution.

Second, the user initiates interaction with an applet. These interactions are sent to the applet *as events to which the applet must respond.* For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a *keypress* event is generated. ***Applets*** can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

2.9. Applet Life Cycle Methods

There are five applet methods that are called by the applet container from the time the applet is loaded into the browser to the time it's terminated by the browser. These methods correspond to various aspects of an applet's life cycle.

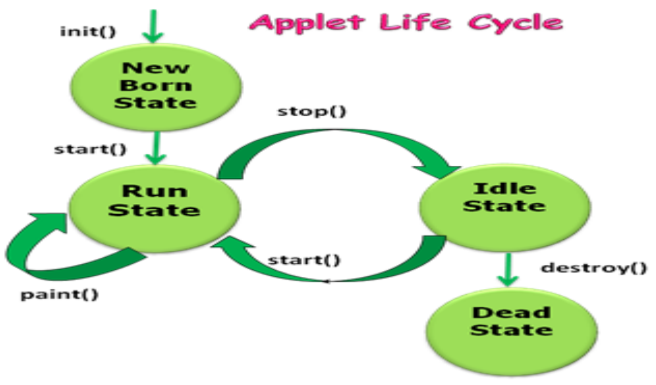


Figure 6: Applet Life Cycle

Methods	**Description**
public void init()	Called once by the ***applet container*** when an applet is loaded for execution. This method initializes an applet. Typical actions performed here are

	initializing fields, creating GUI components, loading sounds to play, loading images to display and creating threads.
public void start()	Called by the ***applet container*** after method ***init*** completes execution. In addition, if the user browses to another website and later returns to the applet's HTML page, method **start** is called again. The method performs any tasks that ***must be completed*** when the applet is loaded for the first time and that must be performed every time the applet's HTML page is revisited. Actions performed here might include starting an animation or starting other threads of execution.
public void paint(Graphics g)	Called by the ***applet container*** after methods ***init*** and ***start***. Method paint is also called when the applet needs to be repainted. For example, if the user covers the applet with another open window on the screen and later uncovers it, the ***paint*** method is called. Typical actions performed here involve drawing with the Graphics object **g** that's passed to the paint method by the applet container.
public void stop()	The ***applet container*** calls this method when the user leaves the applet's web page by browsing to another web page. Since it's possible that the user might return to the web page containing the ***applet***, method ***stop*** performs tasks that might be required to suspend the applet's execution, so that the applet does not use computer processing time when it's not displayed on the screen. Typical actions performed here would stop the execution of animations and threads.
public void destroy()	The ***applet container*** calls this method when the applet is being removed from memory. This occurs when the user exits the browsing session by closing all the browser windows and may also occur at the browser's discretion when the user has browsed to other web pages. The method performs any tasks that are required to ***clean up*** resources allocated to the applet.

2.10.An Applet Skeleton

Four of these methods, **init( ), start( ), stop( ),** and **destroy( ),** apply to all applets and are defined by Applet. Default implementations for all of these methods are provided.

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
        // Called first.
        public void init() {
        // initialization
        }
/* Called second, after init(). Also called whenever the applet is restarted. */
        public void start() {
        // start or resume execution
        }
        // Called when the applet is stopped.
        public void stop() {
        // suspends execution
        }
        /* Called when applet is terminated. This is the last method executed. */
        public void destroy() {
        // perform shutdown activities
        }
        // Called when an applet's window must be restored.
        public void paint(Graphics g) {
        // redisplay contents of window
        }
}
```

2.11.The HTML APPLET Tag

The APPLET tag be used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional while ***code***, ***width*** and ***height*** attributes are required.

```
<APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels
HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
</APPLET>
```

- **CODEBASE** is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file. The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.
- **CODE** is a required attribute that gives the name of the file containing applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
- **ALT** The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets.
- **NAME** is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use ***getApplet( )***, which is defined by the ***AppletContext*** interface.
- **WIDTH** and **HEIGHT** are required attributes that give the size (in pixels) of the applet display area.
- **ALIGN** is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

- **VSPACE** and **HSPACE** are optional attributes. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.
- **PARAM NAME** and **VALUE** The PARAM tag allows us to specify applet-specific arguments in an HTML page. Applets access their attributes with the ***getParameter( )*** method.

2.12.Passing Parameters to Applets

The **APPLET tag** in HTML allows us to pass parameters to our applet. To retrieve a parameter, we need to use the ***getParameter( )*** method. It returns the value of the specified parameter in the form of a ***String*** object. Thus, for ***numeric*** and ***boolean*** values, we will need to convert their string representations into their internal formats.

```
// Use Parameters
/*
<applet code="ParamDemo" width="400" height="100">
<param name="fontName" value="Times new Roman">
<param name="fontSize" value="15">
<param name="leading" value="3">
<param name="accountEnabled" value="true">
</applet>
*/

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

public class ParamDemo extends Applet{
String fontName;
int fontSize;
float leading;
boolean active;
// Initialize the string to be displayed.
public void start() {
     String param;
     fontName = getParameter("fontName");
     if(fontName == null)
     fontName = "Not Found";
     param = getParameter("fontSize");
     try {
          if(param != null) // if not found
          fontSize = Integer.parseInt(param);
          else
              fontSize =0;
     } catch(NumberFormatException e) {
     fontSize = -1;
     }
     param = getParameter("leading");
     try {
     if(param != null) // if not found
     leading = Float.valueOf(param).floatValue();
     else
     leading = 0;
     } catch(NumberFormatException e) {
     leading = -1;
     }
     param = getParameter("accountEnabled");
```

```
      if(param != null)
      active = Boolean.valueOf(param).booleanValue();
      }
// Display parameters.
public void paint(Graphics g) {
      g.drawString("Font name: " + fontName, 0, 10);
      g.drawString("Font size: " + fontSize, 0, 26);
      g.drawString("Leading: " + leading, 0, 42);
      g.drawString("Account Active: " + active, 0, 58);
}
}
```

2.13.Applet Security Basics

Because applets are designed to be loaded from a remote site and then executed locally, security becomes vital. If a user enables Java in the browser, the browser will download all the applet code on the web page and execute it immediately. The user never gets a chance to confirm or to stop individual applets from running. For this reason, applets (unlike applications) are restricted in what they can do. The applet security manager throws a ***SecurityException*** whenever an applet attempts to violate one of the access rules. What can applets do on all platforms? They can show images and *play sounds, get keystrokes and mouse clicks from the user, and send user input back to the host* from which they were loaded. That is enough functionality to show facts and figures or to get user input for placing an order. The restricted execution environment for applets is often called the “sandbox." Applets playing in the “sandbox” cannot alter the user's system or spy on it.

In particular, when running in the sandbox:

- Applets can never run any local executable program.
- Applets cannot communicate with any host other than the server from which they were downloaded; that server is called the ***originating host***. This rule is often called “applets can only phone home.” This protects applet users from applets that might try to spy on intranet resources.
- Applets cannot read from or write to the local computer's file system.
- Applets cannot find out any information about the local computer, except for the Java version used, the name and version of the operating system, and the characters used to separate files (for instance, / or \), paths (such as : or ;), and lines (such as *\n or \r\n*). In particular, applets cannot find out the user's name, e-mail address, and so on.
- All windows that an applet pops up carry a warning message.

All this is possible only because applets are interpreted by the Java Virtual Machine and not directly executed by the CPU on the user's computer. Because the interpreter checks all critical instructions and program areas, a hostile (or poorly written) applet will almost certainly not be able to crash the computer, overwrite system memory, or change the privileges granted by the operating system.

These restrictions are too strong for some situations. For example, on a corporate intranet, we can certainly imagine an applet wanting to access local files. To allow for different levels of security under different situations, we can use ***signed applets***. A signed applet carries with it a certificate that indicates the identity of the signer. Cryptographic techniques ensure that such a certificate cannot be forged. If we trust the signer, we can choose to give the applet additional rights.

The point is that if we trust the signer of the applet, we can tell the browser to give the applet more privileges. The ***configurable Java security model*** allows the continuum of privilege levels we need. We can give completely trusted applets the same privilege levels as local applications. Programs from

vendors that are known to be somewhat flaky can be given access to some, but not all, privileges. Unknown applets can be restricted to the sandbox.
To sum up, Java has three separate mechanisms for enforcing security:

1. Program code is interpreted by the Java Virtual Machine, not executed directly.
2. A security manager checks all sensitive operations in the Java runtime library.
3. Applets can be signed to identify their origin.

2.14. Applet vs Application

Features	**Application**	**Applet**
main() method	Present	Not present
Execution	Requires JRE	Requires a browser or applet viewer
Nature	Called as stand-alone application as application can be executed from command prompt	Requires some third party tool help like a browser to execute
Restrictions	Can access any data or software available on the system	cannot access any thing on the system except browser's services
Security	Does not require any security	Requires highest security for the system as they are untrusted

Table 11: Applet vs Application

2.15. Advantages of Applet

- Execution of applets is easy in a Web browser and does not require any installation or deployment procedure in real-time programming (where as servlets require).
- Writing and displaying (just opening in a browser) graphics and animations is easier than applications.
- In GUI development, constructor, size of frame, window closing code etc. are not required (but are required in applications).

2.16. Application Conversion to Applets

It is easy to convert a graphical Java application that is an application that uses the AWT and that we can start with the java program launcher into an applet that we can embed in a web page. Here are the specific steps for converting an application to an applet.
Make an HTML page with the appropriate tag to load the applet code.

- Supply a subclass of the *JApplet* class. Make this class public. Otherwise, the applet cannot be loaded.
- Eliminate the *main* method in the application. Do not construct a frame window for the application. Our application will be displayed inside the browser.
- Move any initialization code from the frame window constructor to the init method of the applet. We don't need to explicitly construct the applet object. The browser instantiates it for us and calls the *init* method.
- Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.

- Remove the call to setDefaultCloseOperation. An applet cannot be closed; it terminates when the browser exits.
- If the application calls setTitle, eliminate the call to the method. Applets cannot have title bars. (We can, of course, title the web page itself, using the HTML title tag).
- Don't call setVisible(true). The applet is displayed automatically.

Displaying Images

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within
the applet, we use the drawImage method found in the *java.awt.Graphics* class. Following is the example showing all the steps to show images:

```
import  java.applet.*;
import  java.awt.*;
import  java.net.*;
public class ImageDemo extends Applet
{
private Image  image;
private AppletContext context;
public void  init()
{
context = this.getAppletContext();
String  imageURL = this.getParameter("image");
if(imageURL == null)
{
imageURL = "java.jpg";
}
try
{
     URL url=new URL(this.getDocumentBase(),imageURL);
     image =  context.getImage(url);
}catch(MalformedURLException  e)
{
     e. printStackTrace();
// Display in browser status bar
     context.showStatus("Could not load image!");
}
}
public void  paint(Graphics  g)
{
context.showStatus("Displaying image");
     g.drawImage( image, 0, 0, 200, 84, null);
     g.drawString("www.appletdemos.com", 35, 100);
}
}
```

Now, let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="300" height="200">
<param nam e="image" value="java.jpg">
</applet>
<hr>
</html>
```

Playing Audio

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including:
public void play: Plays the audio clip one time, from the beginning.
public void loop: Causes the audio clip to replay continually.
public void stop: Stops playing the audio clip.
To obtain an AudioClip object, we must invoke the getAudioClip method of the Applet class. The getAudioClip method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.
Following is the example showing all the steps to play an audio:

```
import  java. applet.* ;
import  java. awt.* ;
import  java. net.* ;
public class AudioDemo extends Applet
{
private AudioClip  clip;
private AppletContext  context;
public void  init()
{
context = this.getAppletContext();
String  audioURL = this.getParameter("audio");
if( audioURL == null)
{
          audioURL = "default.au";
}
try
{
          URL url=new URL(this.getDocumentBase(),audioURL);
          clip=context.getAudioClip(url);
}catch(MalformedURLException  e)
{
          e.printStackTrace();
          context.showStatus("Could not load audio file!");
}
}
public void  start()
{
if( clip != null)
{
          clip.loop();
}
}
public void  stop()
{
if( clip != null)
{
     clip.stop();
}
}
}
```

Now, let us call this applet as follows:

```
<html>
<title>The AudioDemo applet</title>
<hr>
<applet code="AudioDemo.class" width="0" height="0">
<param name="audio" value="test.wav">
</applet>
```

```
<hr>
</html>
```

2.17. Converting a Java applet to an application

1. Build a Java **JApplet** (or Applet) in Eclipse or NetBeans. (Let's assume you name the class **AppletToApplication**).
2. Change the name of your applet's **init()** method to be the same as the name of your class (in this case: **AppletToApplication**). This is now the constructor method for the class.
3. Delete the word **void** in the header of your new **AppletToApplication** constructor, since a constructor has no return type.
4. Alter the class header so that it extends **Frame** rather than **Applet** (or **JApplet**) .
5. Create a new method called **main**. The header for this method will be:
   public static void main (String[] args)
   This method should create a Frame object as an instance of the class. So, if your class is named **AppletToApplication**, the main method should look like the following (where the size will be your original applet size):

```
public static void main(String[] args)
{
AppletToApplication f = new AppletToApplication ();
f.setSize(300,200);
f.setVisible(true);
f.setLayout(new FlowLayout());
}
```

6. Delete the **import** for the class **Applet** , since it is now redundant.
7. Add window methods (e.g., **windowClosing** to handle the event which is the user clicking on the close window button, and others). This also involves adding **implements WindowListener** and **this.addWindowListener(this)**; to the new constructor method you created in Step #2 above - in order to register the event handler.

```
public void windowClosing(WindowEvent e)
{
dispose();
System.exit(0);
}
public void windowOpened(WindowEvent e)
{ }
public void windowIconified(WindowEvent e)
{ }
public void windowClosed(WindowEvent e)
{ }
public void windowDeiconified(WindowEvent e)
{ }
public void windowActivated(WindowEvent e)
{ }
public void windowDeactivated(WindowEvent e)
{ }
```

8. Make sure that the program does not use any of the methods that are special to the Applet class – methods including **getAudioClip** , **getCodeBase** , **getDocumentBase** , and **getImage**.

Chapter 5: Events, Handling Events and AWT/Swing

5.1. Event Handling Fundamental

Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks. The operating environment reports these events to the programs that are running. Each program then decides what, if anything, to do in response to these events. Event handling is fundamental to Java programming because it is integral to the creation of applets and other types of GUI-based programs. Any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Events are supported by a number of packages, including ***java.util***, ***java.awt***, and ***java.awt.event***.
Most events to which our program will respond are generated when the user interacts with a ***GUI-based program***. They are passed to our program in a variety of ways, with the specific method dependent upon the actual event. There are several types of events, including those generated by the ***mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.***
Events are also occasionally used for purposes not directly related to GUI-based programs. In all cases, the same basic event handling techniques apply.

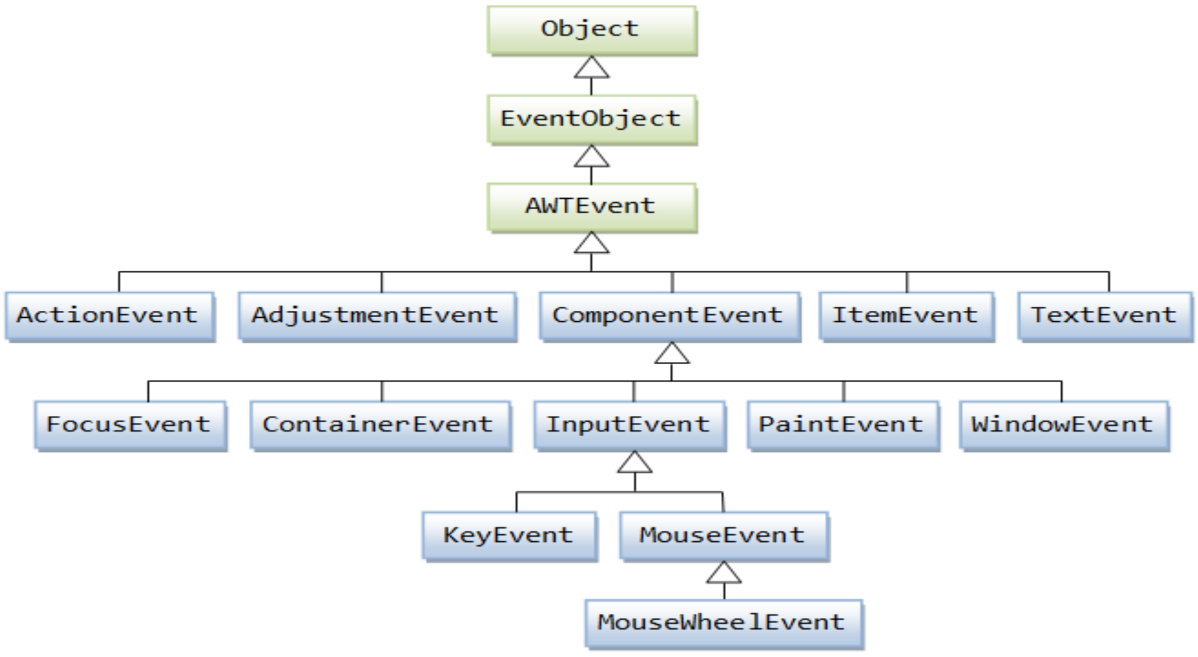


Figure 7: AWT Event Class Diagram

5.2. Events

An event is an *object* that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are *pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse*.

5.2.1. Types of Event

The events can be broadly classified into two categories:
Foreground Events - Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, *clicking on a button, moving the mouse, entering a character through keyboard,*

selecting an item from list, scrolling the page etc.
Background Events - Those events that require the interaction of end user are known as background events. *Operating system interrupts, hardware or software failure, timer expires, an operation completion* are the example of background events.

5.2.2. What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

The Delegation Event Model has the following key participants namely:
Source: The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provide as with classes for source object.
Listener: It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view, the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event an then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model, Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listeners that want to receive them.

5.2.2.1. Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

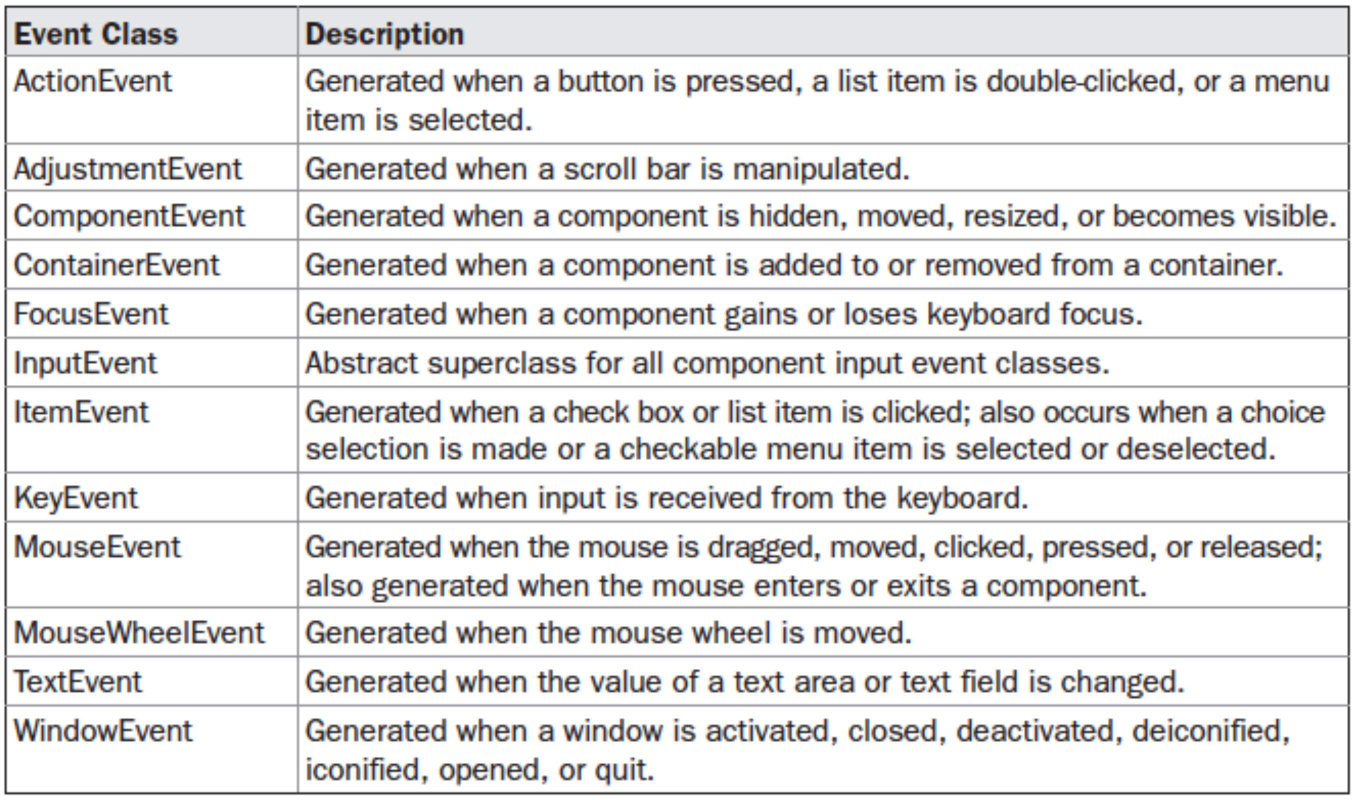
Points to remember about listener
In order to design a listener class, we have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
If we do not implement the any if the predefined interfaces, then our class cannot act as a listener class for a source object.

Callback Methods
These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods

are provided in listener interfaces. If a component wants some listener will listen to its events the the source must register itself to the listener.

5.3. Event Classes

The classes that represent events are at the core of Java's *event handling mechanism*. Thus, a discussion of event handling must begin with the event classes. The most widely used events are those defined by the AWT and those defined by Swing.
At the root of the Java event class hierarchy is *EventObject*, which is in *java.util.* It is the superclass for all events.
EventObject is a superclass of all events.
AWTEvent is a superclass of all AWT events that are handled by the delegation event model.
The package *java.awt.event* defines many types of events that are generated by various user interface elements. Following table shows several commonly used event classes and provides a brief description of when they are generated.



Event Class	**Description**
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 12: Main Event Classes in java.awt.event

5.3.1. The ActionEvent Class

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. ActionEvent class contains the following methods:
String getActionCommand() – Used to obtain the command name for the invoking ActionEvent object.
int getModifiers() – This method returns an integer that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed.
long getWhen() – Returns the time when the event was generated.

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

5.3.2. The AdjustmentEvent Class

An ***AdjustmentEvent*** is generated by a scroll bar. There are five types of adjustment events. The *AdjustmentEvent* class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

The *getAdjustable( )* method returns the object that generated the event. Its form is shown here:
The type of the adjustment event may be obtained by the *getAdjustmentType( )* method. It returns one of the constants defined by *AdjustmentEvent*.
The amount of the adjustment can be obtained from the *getValue( )* method.

5.3.3. The ComponentEvent Class

A *ComponentEvent* is generated when the *size, position, or visibility* of a component is changed. There are four types of component events. The *ComponentEvent* class defines integer constants that can be used to identify them.

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent has this constructor:
ComponentEvent(Component src, int type)
Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.
ComponentEvent is the superclass either directly or indirectly of ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent.
The *getComponent( )* method returns the component that generated the event.
Component getComponent( )

5.3.4. The ContainerEvent Class

A *ContainerEvent* is generated when a component is added to or removed from a container.
There are two types of container events. The *ContainerEvent* class defines int constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED. They indicate that a component has been added to or removed from the container.
ContainerEvent is a subclass of *ComponentEvent* and has this constructor:
Following are the methods available in the *ContainerEvent* class:
Container getContainer() – Returns the reference of the container that generated the event.
Component getChild() – Returns the reference of the component that is added to or removed from the container.

5.3.5. The FocusEvent Class

A *FocusEvent* is generated when a component gains or loses input focus. These events are identified by the integer constants FOCUS_GAINED and FOCUS_LOST.
FocusEvent is a subclass of *ComponentEvent* and has these constructors:
Following are the methods available in the FocusEvent class:
Component getOppositeComponent() – Returns the other component that gained or lost focus.
boolean isTemporary() – Method returns true or false that indicates whether the change in focus is temporary or not.

5.3.6. The InputEvent Class

The abstract class *InputEvent* is a subclass of *ComponentEvent* and is the superclass for component input events. Its subclasses are *KeyEvent* and *MouseEvent*.
InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the *InputEvent* class defined the following eight values to represent the modifiers:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

When writing new code, it is recommended that we use the new, extended modifiers rather than the original modifiers.
To test if a modifier was pressed at the time an event is generated, use the *isAltDown( ), isAltGraphDown( ), isControlDown( ), isMetaDown( ), and isShiftDown( )* methods. The forms of these methods are shown here:
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
We can obtain a value that contains all of the original modifier flags by calling the *getModifiers( )* method.
int getModifiers( )
We can obtain the extended modifiers by calling *getModifiersEx( ),* which is shown here:
int getModifiersEx( )

5.3.7. The ItemEvent Class

An *ItemEvent* is generated when a *check box or a list item is clicked or when a checkable menu item is selected or deselected*. There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

In addition, *ItemEvent* defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state. *ItemEvent* has this constructor:
Following are the methods available in the ItemEvent class:
Object getItem() – Returns a reference to the item whose state has changed.
ItemSelectable getItemSelectable() – Returns the reference of *ItemSelectable*object that raised the event.
int getStateChange() – Returns the status of the state (whether SELECTED or DESELECTED)

5.3.8. The KeyEvent Class

A *KeyEvent* is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASED, and KEY_TYPED. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated.
There are many other integer constants that are defined by *KeyEvent*. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The VK constants specify virtual key codes and are independent of any modifiers, such as *control, shift, or alt*. *KeyEvent* is a subclass of *InputEvent*. KeyEvent is a sub class of InputEvent.
Following are the methods available in KeyEvent class:
char getKeyChar() – Returns the character entered from the keyboard
int getKeyCode() – Returns the key code

5.3.9. The MouseEvent Class

There are eight types of mouse events. The *MouseEvent* class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

MouseEvent is a subclass of *InputEvent*. Here is one of its constructors:
Following are some of the methods available in MouseEvent class:
int getX() – Returns the x-coordinate of the mouse at which the event was generated.
int getY() – Returns the y-coordinate of the mouse at which the event was generated.
Point getPoint() – Returns the x and y coordinates of mouse at which the event was generated.
int getClickCount() – This method returns the number of mouse clicks for an event.
int getButton() – Returns an integer that represents the buttons that caused the event. Returned value can be either NOBUTTON, BUTTON1 (left mouse button), BUTTON2 (middle mouse button), or BUTTON3 (right mouse button).

Following methods are available to obtain the coordinates relative to the screen:
Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()

5.3.10. The MouseWheelEvent Class

The *MouseWheelEvent* class encapsulates a mouse wheel event. It is a subclass of *MouseEvent*. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. *MouseWheelEvent* defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Here is one of the constructors defined by *MouseWheelEvent*:
MouseWheelEvent(Component src, int type, long when, int modifiers,
int x, int y, int clicks, boolean triggersPopup,
int scrollHow, int amount, int count)
Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in x and y. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL. The number of units to scroll is passed in amount. The count parameter indicates the number of rotational units that the wheel moved.

5.3.11. The TextEvent Class

Instances of this class describe text events. These are generated by *text fields and text areas* when characters are entered by a user or program. *TextEvent* defines the integer constant TEXT_VALUE_CHANGED.
The one constructor for this class is shown here:
TextEvent(Object src, int type)
Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.
The *TextEvent* object does not include the characters currently in the text component that generated the *event*. Instead, our program must use other methods associated with the text component to retrieve that information.

5.3.12. The WindowEvent Class

A window event is generated when a user performs any one of the window related events which are identified by the following constants as shown in table:

Following are some of the methods available in the WindowEvent class:
Window getWindow() – Returns the reference of the window on which the event is generated.
Window getOppositeWindow() – Returns the reference to the previous window when a focus event of activation event has occurred.
int getOldState() – Returns an integer indicating the old state of the window.
int getNewState() – Returns an integer indicating the new state of the window.

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

5.4. Sources of Events

Following table lists some of the user interface components that can generate the events. In addition to these graphical user interface elements, any class derived from Component, such as Applet, can generate events. For example, we can receive key and mouse events from an applet.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table: Sources of Events

5.5. Event Listener Interfaces

The delegation event model has two parts: *sources and listeners*. Listeners are created by implementing one or more of the interfaces defined by the *java.awt.event* package. When an event occurs, the event source invokes the appropriate method defined by the *listener* and provides *an event object* as its argument. Below table lists commonly used *listener* interfaces and provides a brief description of the methods that they define.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table: Commonly Used Event Listener Interfaces

5.6. Handling Mouse Event

Handling mouse event using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

To handle mouse events, we must implement the *MouseListener* and the *MouseMotionListener* interfaces.
The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, *mouseX* and *mouseY*, store the location of the mouse when a *mouse pressed, released, or dragged event occurs*. These coordinates are then used by *paint( )* to display output at the point of these occurrences.

```
// Demonstrate the mouse event handlers.
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public  class  MouseEvents  extends  Applet  implements  MouseListener,
MouseMotionListener {
      String msg = "";
      int mouseX = 0, mouseY = 0; // coordinates of mouse
      public void init() {
            addMouseListener(this);
            addMouseMotionListener(this);
      }
      // Handle mouse clicked.
      public void mouseClicked(MouseEvent me) {
            // save coordinates
            mouseX = 0;
            mouseY = 10;
            msg = "Mouse clicked.";
            repaint();
      }
      // Handle mouse entered.
      public void mouseEntered(MouseEvent me) {
            // save coordinates
            mouseX = 0;
            mouseY = 10;
            msg = "Mouse entered.";
            repaint();
      }
      // Handle mouse exited.
      public void mouseExited(MouseEvent me) {
            // save coordinates
            mouseX = 0;
            mouseY = 10;
            msg = "Mouse exited.";
            repaint();
      }
      // Handle button pressed.
      public void mousePressed(MouseEvent me) {
            // save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "Down";
            repaint();
      }
      // Handle button released.
      public void mouseReleased(MouseEvent me) {
            // save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "Up";
            repaint();
      }
      // Handle mouse dragged.
      public void mouseDragged(MouseEvent me) {
            // save coordinates
            mouseX = me.getX();
            mouseY = me.getY();
            msg = "*";
            showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
            repaint();
      }
      // Handle mouse moved.
      public void mouseMoved(MouseEvent me) {
            // show status
            showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
      }
      // Display msg in applet window at current X,Y location.
      public void paint(Graphics g) {
            g.drawString(msg, mouseX, mouseY);
      }
}
```

The ***MouseEvents*** class extends ***Applet*** and implements both the ***MouseListener*** and ***MouseMotionListener*** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the ***applet*** is both the source and the listener for these events. This works because Component, which supplies the ***addMouseListener()*** and ***addMouseMotionListener()*** methods, is a superclass of ***Applet***. Being both the source and the listener for events is a common situation for applets. Inside ***init( ),*** the applet registers itself as a listener for mouse events. This is done by using ***addMouseListener( )*** and ***addMouseMotionListener( ),*** which, are the members of Component.

5.7. Swing

Swing API is set of extensible GUI Components to ease developer's life to create JAVA based Front
End/ GUI Applications. It is built upon top of AWT API and acts as replacement of AWT API as it has almost every control corresponding to AWT controls. Swing component follows a *Model-View-*
Controller architecture to fulfill the following criteria.

- A single API is to be sufficient to support multiple look and feel.
- API is to model driven so that highest level API is not required to have the data.
- API is to use the Java Bean model so that Builder Tools and IDE can provide better services to the developers to use it.

MVC Architecture

Swing API architecture follows loosely based MVC architecture in the following manner.

- A Model represents component's data.
- View represents visual representation of the component's data.
- Controller takes the input from the user on the view and reflects the changes in Component's data.
- Swing component have Model as a separate element and View and Controller part are clubbed in User Interface elements. Using this way, Swing has pluggable look-and-feel architecture.

As a programmer using Swing components, we generally don't need to think about the model-view-controller architecture. Each user interface has a *wrapper class* (such as **JButton** or **JTextField**) that stores the model and the view. When we want to inquire about the contents (for example, the text in a text field), *the wrapper class asks the model and returns the answer to us*. When we want to change the view (for example, move the caret position in a text field), the *wrapper class forwards that request to the view*. However, there are occasions where the wrapper class doesn't work hard enough on forwarding commands. Then, we have to ask it to retrieve the model and work directly with the model. (We don't have to work directly with the view-that is the job of the look-and-feel code.)

Besides being "*the right thing to do*," the ***model-view-controller*** pattern was attractive for the Swing designers because it allowed them to implement pluggable look and feel. The model of a button or text field is independent of the look-and-feel. But of course the visual representation is completely dependent on the user interface design of a particular look and feel. The controller can vary as well.

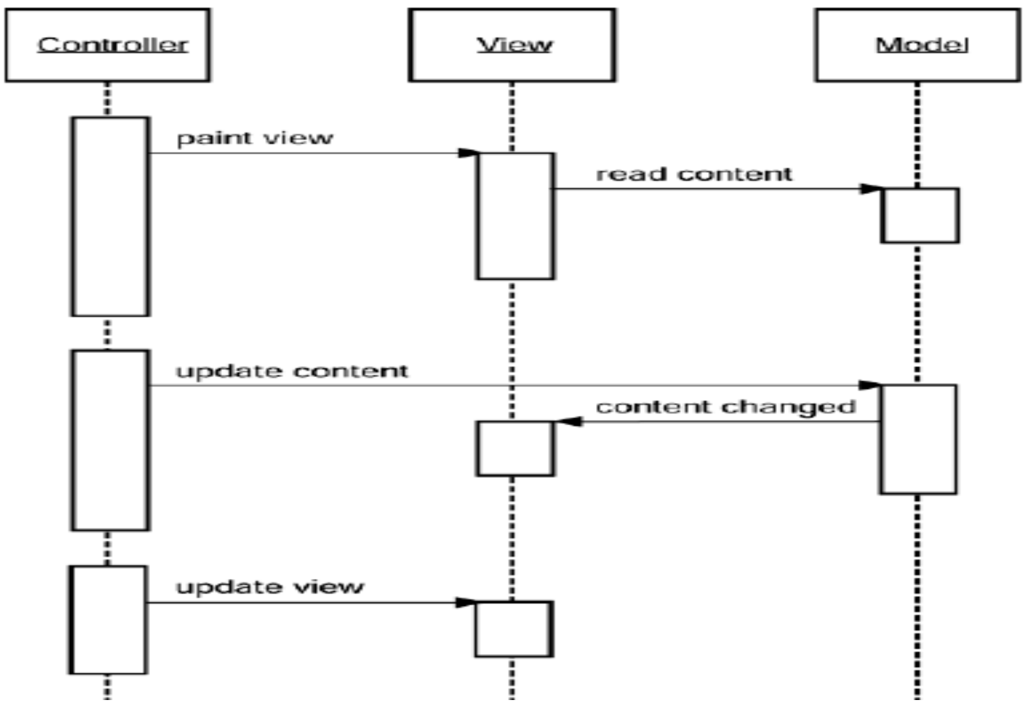


Figure 8: Interaction Between Model, View and Controller objects

Swing Features

- **Light Weight** - Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich controls** - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, table controls.
- **Highly Customizable** - Swing controls can be customized in very easy way as visual appearance is independent of internal representation.
- **Pluggable look and feel**- SWING based GUI Application look and feel can be changed at run time based on available values.

5.7.1. Java Swing VS Java AWT

Java AWT	Java Swing
AWT components are **platform-dependent**.	Java swing components are **platform-independent.**
AWT components are **heavyweight**.	Swing components are **lightweight**.
AWT doesn't support **pluggable look and feel**.	Swing supports **pluggable look and feel**.

AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT doesn't follows MVC where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC.

Table 13: Java Swing vs Java AWT

5.7.2. Swing Controls

Every user interface considers the following three main aspects:

- UI elements: These are the core visual elements the user eventually sees and interacts with. AWT provides a huge list of widely used and common elements varying from basic to complex.
- Layouts: They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface).
- Behaviour: These are events which occur when the user interacts with UI elements.

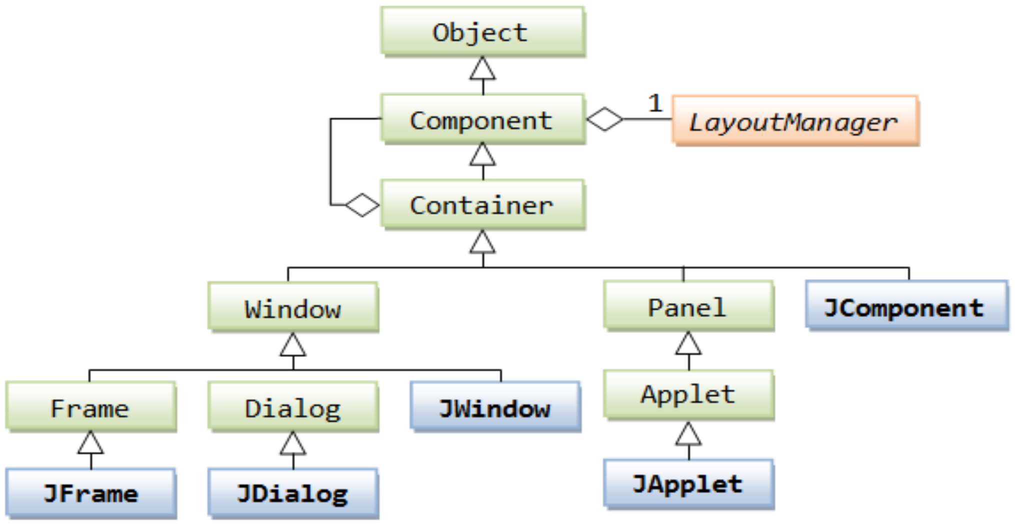


Figure 9: Swing Container Class Diagram

Every SWING controls inherits properties from Component class hierarchy.

1. Component: A Container is the abstract base class for the non-menu user-interface controls of SWING. Component represents an object with graphical representation
2. Container: A Container is a component that can contain other SWING components.
3. JComponent: A JComponent is a base class for all swing UI components. In order to use a swing component that inherits from JComponent, component must be in a containment hierarchy whose root is a top-level Swing container.

Swing UI Elements

Following is the list of commonly used controls while designed GUI using SWING.

SN	Control & Description
1	JLabel: A JLabel object is a component for placing text in a container.
2	JButton: This class creates a labeled button.
3	JColorChooser: A JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.

4	JCheck Box: A JCheckBox is a graphical component that can be in either an on true or off false state.
5	JRadioButton: The JRadioButton class is a graphical component that can be in either an on true or off false state. in a group.
6	JList: A JList component presents the user with a scrolling list of text items.
7	JComboBox: A JComboBox component presents the user with a to show up menu of choices.
8	JTextField: A JTextField object is a text component that allows for the editing of a single line of text.
9	JPasswordField: A JPasswordField object is a text component specialized for password entry.
10	JTextArea: A JTextArea object is a text component that allows for the editing of a multiple lines of text.
11	ImageIcon: A ImageIcon control is an implementation of the Icon interface that paints Icons from Images
12	JScrollbar: A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
13	JOptionPane: JOptionPane provides set of standard dialog boxes that prompt users for a value or informs them of something.
14	JFileChooser: A JFileChooser control represents a dialog window from which the user can select a file.
15	JProgressBar: As the task progresses towards completion, the progress bar displays the task's percentage of completion.
16	JSlider: A JSlider lets the user graphically select a value by sliding a knob within a bounded interval.
17	JSpinner: A JSpinner is a single line input field that lets the user select a number or an object value from an ordered sequence.

Table 14: Swing UIElements

Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Table 15: Commonly used Methods of Component class

Simple Swing Program

```
import javax.swing.*;
public class FirstSwingExample extends JFrame{
    FirstSwingExample()
    {
        JFrame frm=new JFrame("First Swing Example");//creating instance
of JFrame
        JButton btn = new JButton("Click Me");//creating instance of
JButton
        btn.setBounds(130, 100, 100, 40);//x axis, y axis, width, height
        frm.add(btn);//adding button in JFrame
        frm.setSize(400, 500);//400 width and 500 height
        frm.setLayout(null);//using no layout managers
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frm.setVisible(true);//making the frame visible
    }
    public static void main(String[] args) {
        new FirstSwingExample();
    }
}
```

Another Example

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;
public class SwingExample {

    public static void main(String[] args) {
        // Creating instance of JFrame
        JFrame frame = new JFrame("Swing Example");
        // Setting the width and height of frame
        frame.setSize(350, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     /*Creating panel. Inside panels we can add text  fields, buttons
and other components. */
        JPanel panel = new JPanel();
        // adding panel to frame
        frame.add(panel);
/* calling user defined method for adding components to the panel. */
        placeComponents(panel);
        // Setting the frame visibility to true
        frame.setVisible(true);
    }
    private static void placeComponents(JPanel panel) {
        panel.setLayout(null);
        // Creating JLabel
        JLabel userLabel = new JLabel("User");
   /*This method specifies the location and size of component.
setBounds(x, y, width, height) here (x,y) are cordinates from the top
left corner and remaining two arguments are the width and height of the
component. */
        userLabel.setBounds(10,20,80,25);
        panel.add(userLabel);

        /* Creating text field where user is supposed to enter user
name. */
        JTextField userText = new JTextField(20);
        userText.setBounds(100,20,165,25);
        panel.add(userText);
        // Same process for password label and text field.
        JLabel passwordLabel = new JLabel("Password");
        passwordLabel.setBounds(10,50,80,25);
        panel.add(passwordLabel);

        JPasswordField passwordText = new JPasswordField(20);
        passwordText.setBounds(100,50,165,25);
        panel.add(passwordText);

        // Creating login button
        JButton loginButton = new JButton("login");
        loginButton.setBounds(10, 80, 80, 25);
        panel.add(loginButton);
    }

}
```

5.7.3. Swing Event Handling Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SwingControlDemo {
    private JFrame mainFrame;
    private JLabel headerLabel;
    private JLabel statusLabel;
    private JPanel controlPanel;

    public SwingControlDemo() {
        prepareGUI();
    }
    public static void main(String[] args) {
        SwingControlDemo scd = new SwingControlDemo();
        scd.showEventDemo();
    }
    private void prepareGUI() {
        mainFrame = new JFrame("Java SWING Examples");
        mainFrame.setSize(400, 400);
        mainFrame.setLayout(new GridLayout(3, 1));
        headerLabel = new JLabel("", JLabel.CENTER);
        statusLabel = new JLabel("", JLabel.CENTER);
        statusLabel.setSize(350, 100);

        controlPanel = new JPanel();
        controlPanel.setLayout(new FlowLayout());
        mainFrame.add(headerLabel);
        mainFrame.add(controlPanel);
        mainFrame.add(statusLabel);
        mainFrame.setVisible(true);
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    private void showEventDemo() {
        headerLabel.setText("Control in action: Button");
        JButton okButton = new JButton("OK");
        JButton submitButton = new JButton("Submit");
        JButton cancelButton = new JButton("Cancel");
        okButton.setActionCommand("OK");
        submitButton.setActionCommand("Submit");
        cancelButton.setActionCommand("Cancel");
        okButton.addActionListener(new ButtonClickListener());
        submitButton.addActionListener(new ButtonClickListener());
        cancelButton.addActionListener(new ButtonClickListener());
        controlPanel.add(okButton);
        controlPanel.add(submitButton);
        controlPanel.add(cancelButton);
        mainFrame.setVisible(true);
    }
    private class ButtonClickListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            String command = e.getActionCommand();
            if (command.equals("OK")) {
                statusLabel.setText("Ok Button clicked.");
            } else if (command.equals("Submit")) {
                statusLabel.setText("Submit Button clicked.");
            } else {
                statusLabel.setText("Cancel Button clicked.");
            }
        }
    }
}
```

JRadioButton example with event handling

```
import javax.swing.*;
import java.awt.event.*;

class RadioExample extends JFrame implements ActionListener {
    JRadioButton rb1, rb2;
    JButton b;

    RadioExample() {
        rb1 = new JRadioButton("Male");
        rb1.setBounds(100, 50, 100, 30);

        rb2 = new JRadioButton("Female");
        rb2.setBounds(100, 100, 100, 30);

        ButtonGroup bg = new ButtonGroup();
        bg.add(rb1);
        bg.add(rb2);

        b = new JButton("click");
        b.setBounds(100, 150, 80, 30);
        b.addActionListener(this);

        add(rb1);
        add(rb2);
        add(b);

        setSize(300, 300);
        setLayout(null);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (rb1.isSelected()) {
            JOptionPane.showMessageDialog(this, "You are male");
        }
        if (rb2.isSelected()) {
            JOptionPane.showMessageDialog(this, "You are female");
        }
    }
    public static void main(String args[]) {
        new RadioExample();
    }
}
```

5.7.4. A Model-View-Controller Analysis of Swing Buttons

Buttons are about the simplest user interface elements, so they are a good place to become comfortable with the *model-view-controller* pattern. For most components, the model class implements an interface whose name ends in ***Model***. Thus, there is an interface called ***ButtonModel***. Classes implementing that interface can define the state of the various kinds of buttons. Actually, buttons aren't all that complicated, and the Swing library contains a single class, called ***DefaultButtonModel***, that implements this interface.

`getActionCommand()`	The action command string associated with this button
`getMnemonic()`	The keyboard mnemonic for this button
`isArmed()`	`true` if the button was pressed and the mouse is still over the button
`isEnabled()`	`true` if the button is selectable
`isPressed()`	`true` if the button was pressed but the mouse button hasn't yet been released
`isRollover()`	`true` if the mouse is over the button
`isSelected()`	`true` if the button has been toggled on (used for check boxes and radio buttons)

Table 16: The accessor methods of the ButtonModel interface

Each ***JButton*** object stores a button model object, which we can retrieve.

```
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

Building GUI with Swing

The following examples demonstrate the use of ***JButton*** in which whenever we click one of the buttons, the appropriate action listener changes the background color of the panel.

```
import java.awt.*;
 import java.awt.event.*;
 import javax.swing.*;

 public class ButtonTest
 {
      public static void main(String[] args)
      {
           ButtonFrame frame = new ButtonFrame();
           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
          frame.show();
      }
 }

 /**
 A frame with a button panel  */
 class ButtonFrame extends JFrame
 {
     public ButtonFrame()
     {
          setTitle("ButtonTest");
          setSize(WIDTH, HEIGHT);

     // add panel to frame

      ButtonPanel panel = new ButtonPanel();
      Container contentPane = getContentPane();
      contentPane.add(panel);
     }

     public static final int WIDTH = 300;
     public static final int HEIGHT = 200;
 }

 /** A panel with three buttons.  */
 class ButtonPanel extends JPanel
 {
     public ButtonPanel()
     {
           // create buttons

      JButton yellowButton = new JButton("Yellow");
      JButton blueButton = new JButton("Blue");
     JButton redButton = new JButton("Red");

     // add buttons to panel

      add(yellowButton);
      add(blueButton);
      add(redButton);

     // create button actions

     ColorAction yellowAction = new ColorAction(Color.yellow);
     ColorAction blueAction = new ColorAction(Color.blue);
      ColorAction redAction = new ColorAction(Color.red);
     // associate actions with buttons

      yellowButton.addActionListener(yellowAction);
      blueButton.addActionListener(blueAction);
     redButton.addActionListener(redAction);
   }

 /** An action listener that sets the panel's background color. */
      private class ColorAction implements ActionListener
     {
           public ColorAction(Color c)
```

```
            {
                  backgroundColor = c;
            }
            public void actionPerformed(ActionEvent event)
            {
                  setBackground(backgroundColor);
                  repaint();
            }
            private Color backgroundColor;
      }
}
```

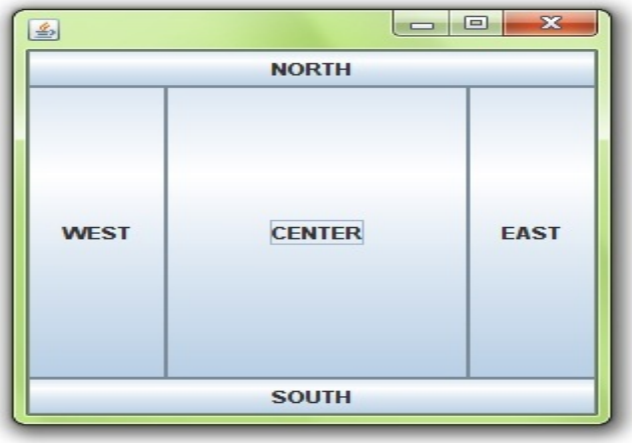
5.7.5. Layout Management

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

1. BorderLayout
2. FlowLayout
3. GridLayout
4. CardLayout
5. GridBagLayout
6. BoxLayout
7. GroupLayout
8. ScrollPaneLayout
9. SpringLayout etc.

5.7.5.1. BorderLayout:

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region: **NORTH ,SOUTH, EAST, WEST, CENTER**



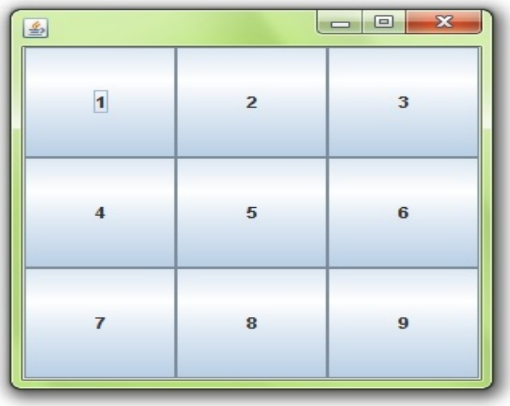
Constructors of BorderLayout class

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

```
import java.awt.*;
import javax.swing.*;

public class Border extends
JFrame {
    JFrame f;
    Border() {
        f = new JFrame();
        JButton b1 = new JButton("NORTH");;
        JButton b2 = new JButton("SOUTH");;
        JButton b3 = new JButton("EAST");;
        JButton b4 = new JButton("WEST");;
        JButton b5 = new JButton("CENTER");;

        f.add(b1, BorderLayout.NORTH);
        f.add(b2, BorderLayout.SOUTH);
        f.add(b3, BorderLayout.EAST);
        f.add(b4, BorderLayout.WEST);
        f.add(b5, BorderLayout.CENTER);
        f.setSize(300, 300);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new Border();
    }
}
```

5.7.5.2. GridLayout

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

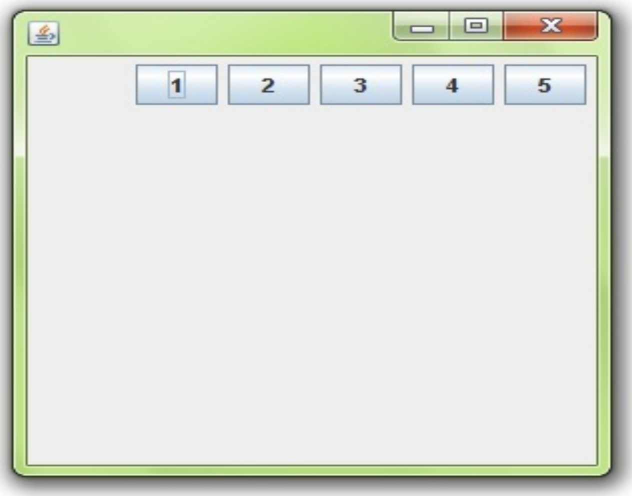


Constructors of GridLayout class:

- **GridLayout():** creates a grid layout with one column per component in a row.
- **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
- **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

```
import java.awt.*;
import javax.swing.*;

public class MyGridLayout {
JFrame f;
MyGridLayout() {
        f = new JFrame();

        JButton b1 = new JButton("1");
        JButton b2 = new JButton("2");
        JButton b3 = new JButton("3");
        JButton b4 = new JButton("4");
        JButton b5 = new JButton("5");
        JButton b6 = new JButton("6");
        JButton b7 = new JButton("7");
        JButton b8 = new JButton("8");
        JButton b9 = new JButton("9");

        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.add(b6);
        f.add(b7);
        f.add(b8);
        f.add(b9);

        f.setLayout(new GridLayout(3, 3));
        //setting grid layout of 3 rows and 3 columns

        f.setSize(300, 300);
        f.setVisible(true);
    }

    public static void main(String[] args) {
        new MyGridLayout();
    }
}
```

5.7.5.3. FlowLayout



The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel. The FlowLayout provides five constants: LEFT, RIGHT, CENTER, LEADING, TRAILING.

Constructors of FlowLayout class:

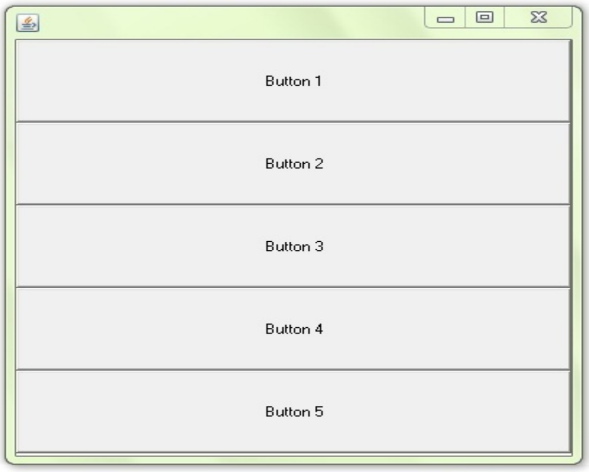
- **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap
- **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap
- **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap

```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout {
    JFrame f;
    MyFlowLayout() {
        f = new JFrame();

        JButton b1 = new JButton("1");
        JButton b2 = new JButton("2");
        JButton b3 = new JButton("3");
        JButton b4 = new JButton("4");
        JButton b5 = new JButton("5");

        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);

        f.setLayout(new FlowLayout(FlowLayout.RIGHT));
        //setting flow layout of right alignment

        f.setSize(300, 300);
        f.setVisible(true);
    }

    public static void main(String[] args) {
        new MyFlowLayout();
    }
}
```

5.7.5.4. BoxLayout class:

The BoxLayout is used to arrange the components either vertically or horizontally. For this purpose, BoxLayout provides four constants. They are as follows: X_AXIS, Y_AXIS, LINE_AXIS, PAGE_AXIS.

Note: BoxLayout class is found in javax.swing package.



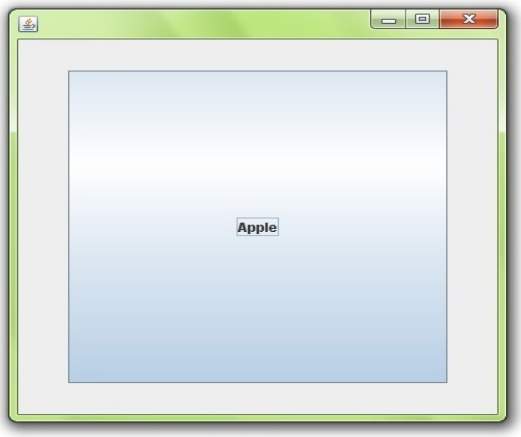
Constructor of BoxLayout class:

- **BoxLayout(Container c, int axis):** creates a box layout that arranges the components with the given axis

```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutExample extends Frame {

    Button buttons[];

    public BoxLayoutExample() {
        buttons = new Button[5];

        for (int i = 0; i < 5; i++) {
            buttons[i] = new Button("Button " + (i + 1));
            add(buttons[i]);
        }
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
        setSize(400, 400);
        setVisible(true);
    }
    public static void main(String args[]) {
        BoxLayoutExample b = new BoxLayoutExample();
    }
}
```

5.7.5.5. CardLayout class

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.



Constructors of CardLayout class

CardLayout(): creates a card layout with zero horizontal and vertical gap.

CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Commonly used methods of CardLayout class:

1. **public void next(Container parent):** is used to flip to the next card of the given container.
2. **public void previous(Container parent):** is used to flip to the previous card of the given container.
3. **public void first(Container parent):** is used to flip to the first card of the given container.
4. **public void last(Container parent):** is used to flip to the last card of the given container.
5. **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

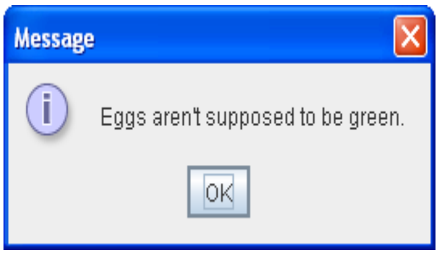
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutExample extends JFrame implements ActionListener
{
    CardLayout card;
    JButton b1, b2, b3;
    Container c;
    CardLayoutExample() {
        c = getContentPane();
        card = new CardLayout(40, 30);
//create CardLayout object with 40 hor space and 30 ver space
        c.setLayout(card);
        b1 = new JButton("Apple");
        b2 = new JButton("Boy");
        b3 = new JButton("Cat");
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        c.add("a", b1);
        c.add("b", b2);
        c.add("c", b3);
    }
    public void actionPerformed(ActionEvent e) {
        card.next(c);
    }
    public static void main(String[] args) {
        CardLayoutExample cl = new CardLayoutExample();
        cl.setSize(400, 400);
        cl.setVisible(true);
        cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

5.7.6. Dialog Box

A Dialog window is an independent sub window meant to carry temporary notice apart from the main Swing Application Window. Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

Several Swing component classes can directly instantiate and display *dialogs*. To create simple, standard dialogs, we use the `JOptionPane` class. The `ProgressMonitor` class can put up a dialog that shows the progress of an operation. Two other classes, `JColorChooser` and `JFileChooser`, also supply standard dialogs.

The code for simple dialogs can be minimal. For example, here is an informational dialog:


Message

Eggs aren't supposed to be green.

Code:

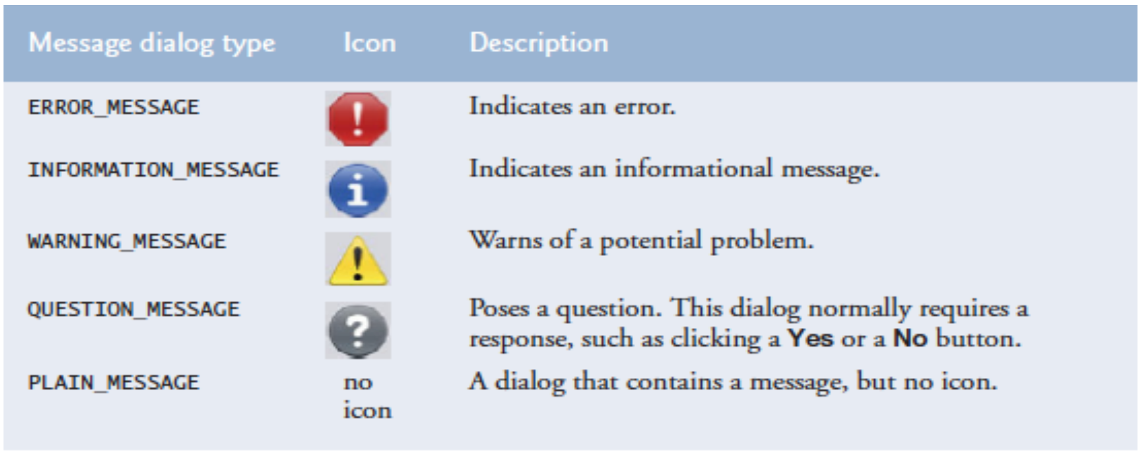
```
JOptionPane.showMessageDialog(frame, "Eggs are
not supposed to be green.");
```

Simple Example program of JOptionPane

```
import javax.swing.JOptionPane;

public class NameDialog
{
   public static void main( String[] args )
   {
      // prompt user to enter name
      String name =
         JOptionPane.showInputDialog( "What is your name?" );
      // create the message
      String message =
         String.format( "Welcome, %s, to Java Programming!", name );

      // display the message to welcome the user by name
      JOptionPane.showMessageDialog( null, message );
   } // end main
} // end class NameDialog
```

Addition Using JOptionPane

```
// Addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane; // program uses JOptionPane
public class Addition
{
   public static void main( String[] args )
   {
      // obtain user input from JOptionPane input dialogs
      String firstNumber =
         JOptionPane.showInputDialog( "Enter first integer" );
      String secondNumber =
          JOptionPane.showInputDialog( "Enter second integer" );
      // convert String inputs to int values for use in a calculation
      int number1 = Integer.parseInt( firstNumber );
      int number2 = Integer.parseInt( secondNumber );
      int sum = number1 + number2; // add numbers
      // display result in a JOptionPane message dialog
      JOptionPane.showMessageDialog( null, "The sum is " + sum,
         "Sum of Two Integers", JOptionPane.PLAIN_MESSAGE );
   } // end method main
} // end class Addition
```



Message dialog type	Icon	Description
ERROR_MESSAGE	❗	Indicates an error.
INFORMATION_MESSAGE	ℹ	Indicates an informational message.
WARNING_MESSAGE	⚠	Warns of a potential problem.
QUESTION_MESSAGE	❓	Poses a question. This dialog normally requires a response, such as clicking a **Yes** or a **No** button.
PLAIN_MESSAGE	no icon	A dialog that contains a message, but no icon.

Table: JOptionPane static constants for message dialogs.

5.7.7.Scroll Bar

A `JScrollPane` provides a scrollable view of a component. When screen real estate is limited, use a scroll pane to display a component that is large or one whose size can change dynamically. Other containers used to save screen space include split panes and tabbed panes.
Here's the code that creates the text area, makes it the scroll pane's client, and adds the scroll pane to a container:

```
//In a container that uses a BorderLayout:
textArea = new JTextArea(5, 30);
...
JScrollPane scrollPane = new JScrollPane(textArea);
...
setPreferredSize(new Dimension(450, 110));
...
add(scrollPane, BorderLayout.CENTER);
```

Additional Program

```
import java.awt.BorderLayout;
import java.awt.GridLayout;

import javax.swing.ButtonGroup;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.JScrollPane;

public class ScrollDemo extends JFrame {

    JScrollPane scrollpane;

    public ScrollDemo() {
        super("JScrollPane Demonstration");
        setSize(300, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        init();
        setVisible(true);
    }
    public void init() {
        JRadioButton form[][] = new JRadioButton[12][5];
        String counts[] = {"", "0-1", "2-5", "6-10", "11-100", "101+"};
        String categories[] = {"Household", "Office", "Extended Family",
            "Company (US)", "Company (World)", "Team", "Will",
            "Birthday Card List", "High School", "Country", "Continent",
            "Planet"};
        JPanel p = new JPanel();
        p.setSize(600, 400);
        p.setLayout(new GridLayout(13, 6, 10, 0));
        for (int row = 0; row < 13; row++) {
            ButtonGroup bg = new ButtonGroup();
            for (int col = 0; col < 6; col++) {
                if (row == 0) {
                    p.add(new JLabel(counts[col]));
                } else {
                    if (col == 0) {
                        p.add(new JLabel(categories[row - 1]));
                    } else {
                        form[row - 1][col - 1] = new JRadioButton();
                        bg.add(form[row - 1][col - 1]);
                        p.add(form[row - 1][col - 1]);
                    }
                }
            }
        }
        scrollpane = new JScrollPane(p);
        getContentPane().add(scrollpane, BorderLayout.CENTER);
    }

    public static void main(String args[]) {
        new ScrollDemo();
    }
}
```

5.7.8. Menus

A menu provides a space-saving way to let the user choose one of several options. Other components with which the user can make a one-of-many choices include *combo boxes*, *lists*, *radio buttons*, *spinners*, and *tool bars*.
Menus are unique in that, by convention, they aren't placed with the other components in the UI. Instead, a menu usually appears either in a *menu bar* or as a *popup menu*. A menu bar contains one or more menus and has a customary, platform-dependent location , usually along the top of a window. A popup menu is a menu that is invisible until the user makes a platform-specific mouse action, such as pressing the right mouse button, over a popup-enabled component. The popup menu then appears under the cursor.

```
import javax.swing.*;
import java.awt.event.*;

public class Notepad implements ActionListener {

    JFrame f;
    JMenuBar mb;
    JMenu file, edit, help;
    JMenuItem cut, copy, paste, selectAll;
    JTextArea ta;

    Notepad() {
        f = new JFrame();

        cut = new JMenuItem("cut");
        copy = new JMenuItem("copy");
        paste = new JMenuItem("paste");
        selectAll = new JMenuItem("selectAll");

        cut.addActionListener(this);
        copy.addActionListener(this);
        paste.addActionListener(this);
        selectAll.addActionListener(this);

        mb = new JMenuBar();
        mb.setBounds(5, 5, 400, 40);

        file = new JMenu("File");
        edit = new JMenu("Edit");
        help = new JMenu("Help");

        edit.add(cut);
        edit.add(copy);
        edit.add(paste);
        edit.add(selectAll);


        mb.add(file);
        mb.add(edit);
        mb.add(help);

        ta = new JTextArea();
        ta.setBounds(5, 30, 460, 460);

        f.add(mb);
        f.add(ta);

        f.setLayout(null);
        f.setSize(500, 500);
        f.setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == cut) {
            ta.cut();
        }
        if (e.getSource() == paste) {
            ta.paste();
        }
        if (e.getSource() == copy) {
            ta.copy();
        }
        if (e.getSource() == selectAll) {
            ta.selectAll();
        }
    }
    public static void main(String[] args) {
        new Notepad();
    }
}
```

5.7.9.Progress Bar

Sometimes a task running within a program might take a while to complete. A user-friendly program provides some indication to the user that the task is occurring, how long the task might take, and how much work has already been done. One way of indicating work, and perhaps the amount of progress, is to use an animated image.

Commonly used Constructors of JProgressBar class

- **JProgressBar():** is used to create a horizontal progress bar but no string text.
- **JProgressBar(int min, int max):** is used to create a horizontal progress bar with the specified minimum and maximum value.
- **JProgressBar(int orient):** is used to create a progress bar with the specified orientation, it can be either Vertical or Horizontal by using **SwingConstants.VERTICAL** and **SwingConstants.HORIZONTAL** constants.
- **JProgressBar(int orient, int min, int max):** is used to create a progress bar with the specified orientation, minimum and maximum value.

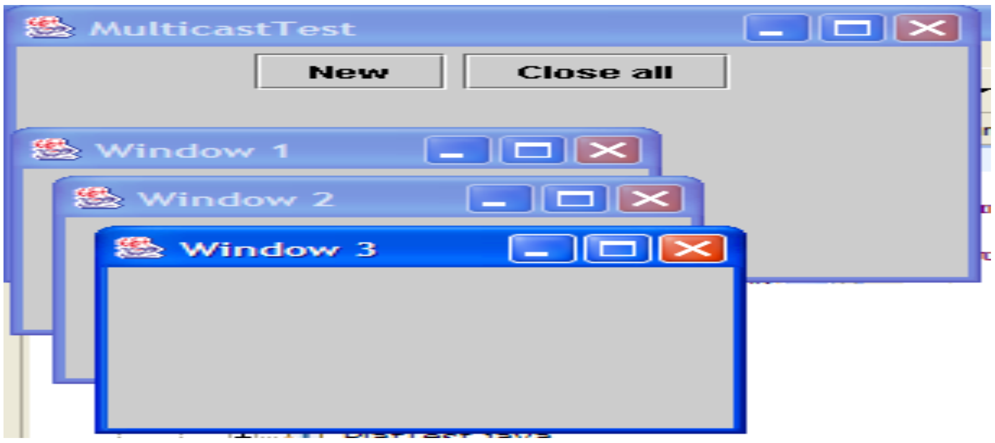
Additional Program

```
import javax.swing.*;
public class MyProgress extends JFrame {
    JProgressBar jb;
    int i = 0, num = 0;

    MyProgress() {
        jb = new JProgressBar(0, 2000);
        jb.setBounds(40, 40, 200, 30);

        jb.setValue(0);
        jb.setStringPainted(true);

        add(jb);
        setSize(400, 400);
        setLayout(null);
    }

    public void iterate() {
        while (i <= 2000) {
            jb.setValue(i);
            i = i + 20;
            try {
                Thread.sleep(150);
            } catch (Exception e) {
            }
        }
    }

    public static void main(String[] args) {
        MyProgress m = new MyProgress();
        m.setVisible(true);
        m.iterate();
    }
}
```

5.8. Multicasting Events

Multicast events are quite useful when we have to work with many windows in our application and we just want to perform the same action or a group of actions in a number of windows in our application at the same time.
For example, if the user has opened many windows in the application and he/she might want to provide a command that closes all the windows at once.
In order to work with multicast events, we have to:

- Create a class that extends **JPanel** and implements ActionListener. This class should have a JButton component as a private member. This will be the button that will give the command to all the windows.
- Override the actionPerformed method of this class to bundle a second button that performs a specific action. In our case the creation of a new window.
- The new windows that will be launched will also implement ActionListener.
- Register the new ActionListeners to the button that gives the command to all the windows. So, when this button is pressed all ActionListeners that are resistered to it will be launched and their actionPerformed method will be executed



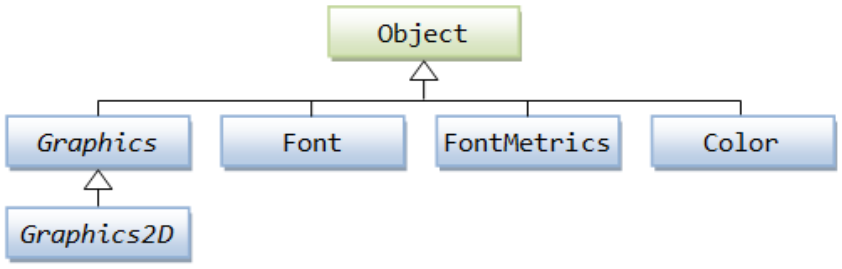
Example Code:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MulticastEvent extends JPanel implements ActionListener {
    private int counter = 0;
    private JButton closeAllButton;
    public MulticastEvent() {
        JButton newButton = new JButton("New");
        add(newButton);
        newButton.addActionListener(this);
        closeAllButton = new JButton("Close all");
        add(closeAllButton);
    }
    public void actionPerformed(ActionEvent evt) {
        CloseFrame f = new CloseFrame();
        counter++;
        f.setTitle("Window " + counter);
        f.setSize(200, 150);
        f.setLocation(30 * counter, 30 * counter);
        f.show();
        closeAllButton.addActionListener(f);
    }
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setTitle("MulticastTest");
        frame.setSize(300, 200);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        Container contentPane = frame.getContentPane();
        contentPane.add(new MulticastEvent());
        frame.show();
    }
    class CloseFrame extends JFrame implements ActionListener {
        public void actionPerformed(ActionEvent evt) { //handles Close
all
            // button
            setVisible(false);
        }
    }
}
```

Chapter 6: Graphics and Images/Animation/Multimedia

6.1. Introduction to Graphics Programming

A *graphics context* provides the capabilities of drawing on the screen. The graphics context maintains states such as the *color* and font used in drawing, as well as interacting with the underlying operating system to perform the drawing. In Java, custom painting is done via the `java.awt.Graphics` class, which manages a graphics context, and provides a set of *device-independent* methods for drawing *texts*, *figures* and *images* on the screen on different platforms. The `java.awt.Graphics` is an `abstract` class, as the actual act of drawing is system-dependent and device-dependent. Each operating platform will provide a subclass of `Graphics` to perform the actual drawing un `Class`

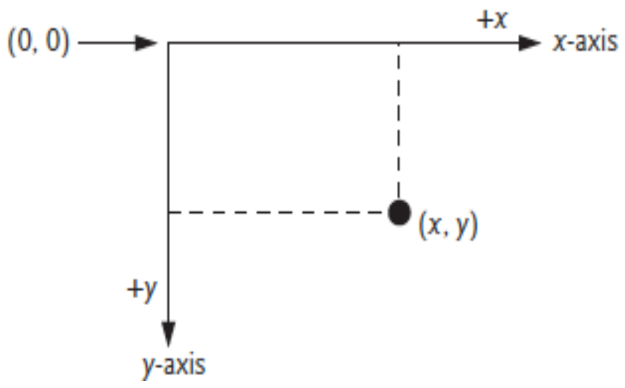


Color contains methods and constants for manipulating colors. Class *JComponent* contains method paintComponent, which is used to draw graphics on a component. Class *Font* contains methods and constants for manipulating fonts. Class *FontMetrics* contains methods for obtaining font information. Class *Graphics* contains methods for drawing strings, lines, rectangles and other shapes. Class *Graphics2D*, which extends class Graphics, is used for drawing with the Java 2D API.

The Graphics class provides methods for drawing three types of graphical objects:

- Text strings: via the *drawString()* method.
- Vector-graphic primitives and shapes: via methods *drawXxx()* and *fillXxx(),* where *Xxx* could be *Line*, *Rect*, *Oval*, *Arc*, *PolyLine*, *RoundRect*, or *3DRect*.
- Bitmap images: via the *drawImage()* method.

In Java's coordinate system the upper-left corner of a GUI component (e.g., a window) has the coordinates (0, 0). A coordinate pair is composed of an x-coordinate and a y-coordinate.The x-coordinate is the horizontal distance moving *right from the left* of the screen. The y-coordinate is the vertical distance moving *down from the top* of the screen. The x-axis describes every horizontal coordinate, and the y-axis every vertical coordinate. The coordinates are used to indicate where graphics should be displayed on a screen. Coordinate units are measured in pixels (which stands for "picture element"). A pixel is a display monitor's smallest unit of resolution.



6.2. Creating Closable Frame

There are a number of techniques that one can use to close a frame in Java. Each has its advantages and disadvantages. Three techniques are presented below in general decreasing preference order:

6.2.1. JFrame.defaultCloseOperation(int option)

This is the simplest technique. All we need to do is to call the `defaultClostOperation` method of a `JFrame`, supplying the desired option value.

The possibilities for the option value are:

- **`JFrame.EXIT_ON_CLOSE`**: A `System.exit(0) call will be executed, exiting the entire application.`
- **`JFrame.DISPOSE_ON_CLOSE`** `-- The frame will be closed and disposed but the application will not exit.`
- **`JFrame.DO_NOTHING_ON_CLOSE`** `-- The frame will be closed but not disposed and the application will not exit.`

The advantage of the technique is that it is very simple and easy, but it does not allow much flexibility to do any custom operations during the frame closing.

Example:

```
public class MyFrame extends JFrame {
    public MyFrame() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    ...
}
```

Note: For dynamically configurable frame closing behaviour, supply the option value to the constructor of the frame. This would be done by the controller, which would instantiate the frame with either an `EXIT_ON_CLOSE` option if it was an application or a `DO_NOTHING_ON_CLOSE` or `DISPOSE_ON_CLOSE` if it were an applet.

6.2.2. java.awt.Window.addWindowListener(java.awt.event.WindowListener l)

In this technique a `WindowListener` interface implentation is added to the frame, where the listener has a method, `windowClosing()`, that is called when the frame is closed.

In practice, on overrides the `windowClosing()` method of `WindowAdapter`, a no-op implementation of `WindowListener`. This way, one doesn't have to worry about all the other methods of the `WindowListener`.

For example:

```
public class MyFrame extends JFrame {
    public MyFrame() {
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                System.exit(0);
            }
        });
    }
    ...
    }
```

This technique has the advantage that any sort of custom processing can be done when the frame is closed. The frame can be constructed with a command that the `windowClosing` method calls or with an complete `WindowListener` object, thus enabling dynamic configuration of the frame closing behavior. This is very useful when we want to run the same frame as either a regular application or as an applet, the controller will instantiate the frame with the proper closing behaviour.
The disadvantage of this technique is the larger code required than the first technique for standard window closing behaviours.

6.2.3. java.awt.Window.processWindowEvent(java.awt.event.WindowEvent e)

In this technique, the `processWindowEvent` method, which is called when the frame is closed, is overriden to provide the desired behavior. The supplied WindowEvent parameter is tested to see if it is the event for the window is being closed and if so, the desired behavior is executed.
For example:

```
    public class MyFrame extends JFrame {
        public MyFrame() {
            protected void processWindowEvent(WindowEvent e) {
                super.processWindowEvent(e);
                if(e.getID() == WindowEvent.WINDOW_CLOSING) {
                    System.exit(0);
                }
            }
        }
        ...
    }
```

For dynamic behavior, a command supplied to the constructor of the frame can be run when the `WINDOW_CLOSING` event is detected.
The advantage of this technique is that an entire object to handle the frame closing needs to be constructed but at the expense of a more complicated logical process.

6.3. Color Control

Class *Color* declares methods and constants for manipulating colors in a Java program. Several color methods and constructors are as follows:

Method	Description
public Color(int r, int g, int b)	Creates a color based on red, green and blue components expressed as integers from 0 to 255.
public Color(float r, float g, float b)	Creates a color based on red, green and blue components expressed as floating point values from 0.0 to 1.0.
public int getRed()	Returns a value between 0 and 255 representing the red content.
public int getGreen()	Returns a value between 0 and 255 representing the green content.
public int getBlue()	Returns a value between 0 and 255 representing the blue content.
public Color getColor()	Returns Color object representing current color for the graphics context.
public void setColor(Color c)	Sets the current color for drawing with the graphics context.

Table 17: Color methods and constructor

Example Program

```
//Demonstrating Colors. (ColorJPanel.java)
import java.awt.Graphics;
import java.awt.Color;
import javax.swing.JPanel;

public class ColorJPanel extends JPanel
{
   // draw rectangles and Strings in different colors
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // call superclass's paintComponent

      this.setBackground( Color.WHITE );

      // set new drawing color using integers
      g.setColor( new Color( 255, 0, 0 ) );
      g.fillRect( 15, 25, 100, 20 );
      g.drawString( "Current RGB: " + g.getColor(), 130, 40 );

      // set new drawing color using floats
      g.setColor( new Color( 0.50f, 0.75f, 0.0f ) );
      g.fillRect( 15, 50, 100, 20 );
      g.drawString( "Current RGB: " + g.getColor(), 130, 65 );

      // set new drawing color using static Color objects
      g.setColor( Color.BLUE );
      g.fillRect( 15, 75, 100, 20 );
      g.drawString( "Current RGB: " + g.getColor(), 130, 90 );

      // display individual RGB values
      Color color = Color.MAGENTA;
      g.setColor( color );
      g.fillRect( 15, 100, 100, 20 );
      g.drawString( "RGB values: " + color.getRed() + ", " +
         color.getGreen() + ", " + color.getBlue(), 130, 115 );
   } // end method paintComponent
} // end class ColorJPanel

//Demonstrating Colors (ShowColors.java).
import javax.swing.JFrame;

public class ShowColors
{
   // execute application
   public static void main( String[] args )
   {
      // create frame for ColorJPanel
      JFrame frame = new JFrame( "Using colors" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      ColorJPanel colorJPanel = new ColorJPanel(); // create ColorJPanel
      frame.add( colorJPanel ); // add colorJPanel to frame
      frame.setSize( 400, 180 ); // set frame size
      frame.setVisible( true ); // display frame
   } // end main
} // end class ShowColors
```

In above program Graphics method ***fillRect*** to draw a filled rectangle in the current *color*. Method fillRect draws a rectangle based on its four arguments. The first two integer values represent the upper-left x-coordinate and upper-left y-coordinate, where the Graphics object begins drawing the rectangle. The third and fourth arguments are nonnegative integers that represent the width and the height of the rectangle in pixels, respectively. A rectangle drawn using method *fillRect* is filled by the current color of the Graphics object.

6.4. Manipulating Fonts

Most font methods and font constants are part of class *Font*. Some methods of class *Font* and class *Graphics* are summarized as follows.

Methods or Constant	Description
public final static int PLAIN	A constant representing a plain font style.
public final static int BOLD	A constant representing a bold font style.
public final static int ITALIC	A constant representing an italic font style.
public Font(String name, int style, int size)	Creates a Font object with the specified font name, style and size.
public int getStyle()	Returns an int indicating the current font style.
public int getSize()	Returns an int indicating the current font size.
public String getName()	Returns the current font name as a string.
public String getFamily()	Returns the font's family name as a string.
public boolean isPlain()	Returns true if the font is plain, else false.
public boolean isBold()	Returns true if the font is bold, else false.
public boolean isItalic()	Returns true if the font is italic, else false.
public Font getFont()	Returns a Font object reference representing the current font.
public void setFont(Font f)	Sets the current font to the font, style and size specified by the Font object reference f.

Table 18: font methods and font constants

Class Font's constructor takes three arguments—the *font name*, *font style* and *font size*. The font name is any font currently supported by the system on which the program is running, such as standard Java fonts *Monospaced*, *SansSerif* and *Serif*. The font style is **Font.PLAIN**, **Font.ITALIC** or **Font.BOLD**. Font styles can be used in combination (e.g., **Font.ITALIC + Font.BOLD**). The font size is measured in points. A point is 1/72 of an inch. Graphics method setFont sets the current drawing font the font in which text will be displayed to its Font argument.

Example Program

FontJPanel.java

```
// Display strings in different fonts and colors.
import java.awt.Font;
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class FontJPanel extends JPanel
{
   // display Strings in different fonts and colors
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // call superclass's paintComponent

 // set font to Serif (Times), bold, 12pt and draw a string
      g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
      g.drawString( "Serif 12 point bold.", 20, 30 );

 // set font to Monospaced (Courier), italic, 24pt and draw a string
      g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
      g.drawString( "Monospaced 24 point italic.", 20, 50 );

// set font to SansSerif (Helvetica), plain, 14pt and draw a string
      g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
      g.drawString( "SansSerif 14 point plain.", 20, 70 );

// set font to Serif (Times), bold/italic, 18pt and draw a string
      g.setColor( Color.RED );
      g.setFont( new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
      g.drawString( g.getFont().getName() + " " + g.getFont().getSize()
+
         " point bold italic.", 20, 90 );
   } // end method paintComponent
} // end class FontJPanel
```

Fonts.java

```
// Using fonts.
import javax.swing.JFrame;

public class Fonts
{
   // execute application
   public static void main( String[] args )
   {
      // create frame for FontJPanel
      JFrame frame = new JFrame( "Using fonts" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      FontJPanel fontJPanel = new FontJPanel(); // create FontJPanel
      frame.add( fontJPanel ); // add fontJPanel to frame
      frame.setSize( 420, 150 ); // set frame size
      frame.setVisible( true ); // display frame
   } // end main
} // end class Fonts
```

6.4.1. Font Metrics

Sometimes it's necessary to get information about the current drawing font, such as its name, style and size. Method ***getStyle*** returns an integer value representing the current style. The integer value returned is either **Font.PLAIN**, **Font.ITALIC**, **Font.BOLD** or the combination of **Font.ITALIC** and **Font.BOLD**. Method **getFamily** returns the name of the font family to which the current font belongs. The name of the font family is platform specific. **Font** methods are also available to test the style of the current font. Methods **isPlain**, **isBold** and **isItalic** return **true** if the current font style is *plain*, *bold* or *italic*, respectively. Some of the common font metrics, which provide precise information about a font, such as **height**, **descent** (the amount a character dips below the baseline), **ascent** (the amount a character rises above the baseline) and **leading** (the difference between the descent of one line of text and the ascent of the line of text below it that is, the interline spacing).

MetricsJPanel.java

```
// FontMetrics and Graphics methods useful for obtaining font metrics.
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import javax.swing.JPanel;

public class MetricsJPanel extends JPanel
{
   // display font metrics
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // call superclass's paintComponent

      g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
      FontMetrics metrics = g.getFontMetrics();
      g.drawString( "Current font: " + g.getFont(), 10, 30 );
      g.drawString( "Ascent: " + metrics.getAscent(), 10, 45 );
      g.drawString( "Descent: " + metrics.getDescent(), 10, 60 );
      g.drawString( "Height: " + metrics.getHeight(), 10, 75 );
      g.drawString( "Leading: " + metrics.getLeading(), 10, 90 );

      Font font = new Font( "Serif", Font.ITALIC, 14 );
      metrics = g.getFontMetrics( font );
      g.setFont( font );
      g.drawString( "Current font: " + font, 10, 120 );
      g.drawString( "Ascent: " + metrics.getAscent(), 10, 135 );
      g.drawString( "Descent: " + metrics.getDescent(), 10, 150 );
      g.drawString( "Height: " + metrics.getHeight(), 10, 165 );
      g.drawString( "Leading: " + metrics.getLeading(), 10, 180 );
   } // end method paintComponent
} // end class MetricsJPanel

Metrics.java
// Displaying font metrics.
import javax.swing.JFrame;

public class Metrics
{
   // execute application
   public static void main( String[] args )
   {
      // create frame for MetricsJPanel
      JFrame frame = new JFrame( "Demonstrating FontMetrics" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      MetricsJPanel metricsJPanel = new MetricsJPanel();
      frame.add( metricsJPanel ); // add metricsJPanel to frame
      frame.setSize( 510, 240 ); // set frame size
      frame.setVisible( true ); // display frame
   } // end main
} // end class Metrics
```

6.5. Drawing Lines, Rectangles and Ovals

This section presents. The methods and their parameters of **Graphics** methods for drawing *lines, rectangles* and *ovals* are summarized in below table. For each drawing method that requires a **width** and **height** parameter, the **width** and **height** must be nonnegative values. Otherwise, the shape will not display.

Methods	Description
public void drawLine(int x1, int y1, int x2, int y2)	Draws a line between the point (x1, y1) and the point (x2, y2).
public void drawRect(int x, int y, int width, int height)	Draws a rectangle of the specified width and height. The rectangle's top-left corner is located at (x, y). Only the outline of the rectangle is drawn using the Graphics object's color, the body of the rectangle is not filled with this color.
public void fillRect(int x, int y, int width, int height)	Draws a filled rectangle in the current color with the specified width and height. The rectangle's top-left corner is located at (x, y).
public void clearRect(int x, int y, int width, int height)	Clears the specified rectangle by filling it with the background color of the current drawing surface.
public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Draws a rectangle with rounded corners in the current color with the specified width and height. The arcWidth and arcHeight determine the rounding of the corners. Only the outline of the shape is drawn.
public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Draws a filled rectangle in the current color with rounded corners with the specified width and height. The arcWidth and arcHeight determine the rounding of the corners.
public void draw3DRect(int x, int y, int width, int height, boolean b)	Draws a three-dimensional rectangle in the current color with the specified width and height. The rectangle's top-left corner is located at (x, y). The rectangle appears raised when b is true and lowered when b is false. Only the outline of the shape is drawn.
public void fill3DRect(int x, int y, int width, int height, boolean b)	Draws a filled three-dimensional rectangle in the current color with the specified width and height. The rectangle's top-left corner is located at (x, y). The rectangle appears raised when b is true and lowered when b is false.
public void drawOval(int x, int y, int width, int height)	Draws an oval in the current color with the specified width and height. The bounding rectangle's top-left corner is located at (x, y). The oval touches all four sides of the bounding rectangle at the center of each side. Only the outline of the shape is drawn.
public void fillOval(int x, int y, int width, int height)	Draws a filled oval in the current color with the specified width and height. The bounding rectangle's top-left corner is located at (x, y). The oval touches the center of all four sides of the bounding rectangle.

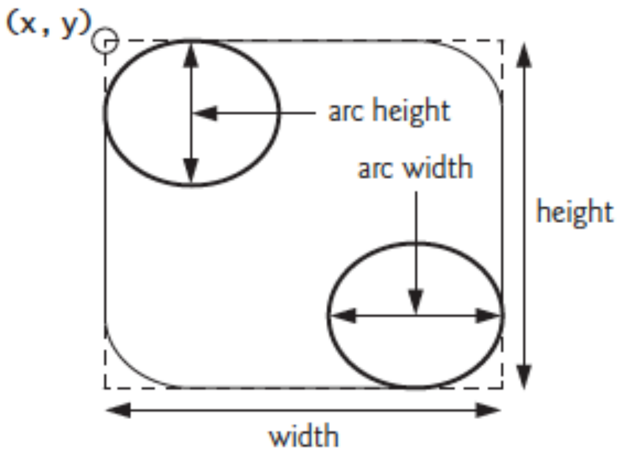


Figure 10: Arc width and arc height for rounded rectangles

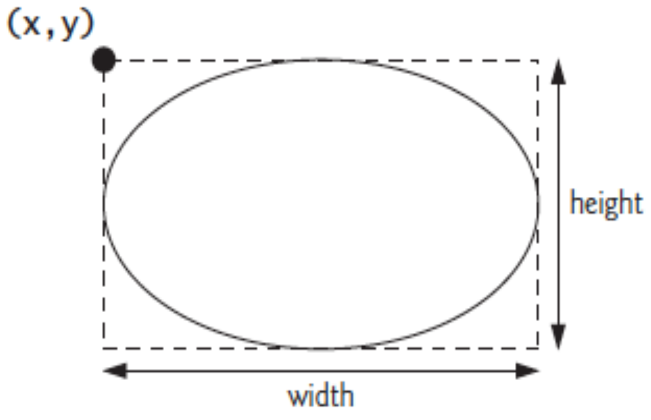


Figure 11: Oval bounded by a rectangle.

LinesRectsOvalsJPanel.java

```
// Drawing lines, rectangles and ovals.
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class LinesRectsOvalsJPanel extends JPanel
{
   // display various lines, rectangles and ovals
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // call superclass's paint method

      this.setBackground( Color.WHITE );

      g.setColor( Color.RED );
      g.drawLine( 5, 30, 380, 30 ); //red line

      g.setColor( Color.BLUE );
      g.drawRect( 5, 40, 90, 55 ); //blue rectangle
      g.fillRect( 100, 40, 90, 55 ); //Blue colored filled rectangle

      g.setColor( Color.BLACK );
      g.fillRoundRect( 195, 40, 90, 55, 50, 50 );//rounded black filled
rectangle
      g.drawRoundRect( 290, 40, 90, 55, 20, 20 );//rounded black
rectangle

      g.setColor( Color.GREEN );
      g.draw3DRect( 5, 100, 90, 55, true ); //green 3D rectangle
      g.fill3DRect( 100, 100, 90, 55, false );//green filled 3D
rectangle

      g.setColor( Color.MAGENTA );
      g.drawOval( 195, 100, 90, 55 ); //magenta colored Oval
      g.fillOval( 290, 100, 90, 55 ); //magenta colored filled Oval
   } // end method paintComponent
} // end class LinesRectsOvalsJPanel
```

LinesRectsOvals.java
```
// Drawing lines, rectangles and ovals.
import java.awt.Color;
import javax.swing.JFrame;

public class LinesRectsOvals
{
   // execute application
   public static void main( String[] args )
   {
      // create frame for LinesRectsOvalsJPanel
      JFrame frame =
         new JFrame( "Drawing lines, rectangles and ovals" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      LinesRectsOvalsJPanel linesRectsOvalsJPanel =
         new LinesRectsOvalsJPanel();
      linesRectsOvalsJPanel.setBackground( Color.WHITE );
      frame.add( linesRectsOvalsJPanel ); // add panel to frame
      frame.setSize( 400, 210 ); // set frame size
      frame.setVisible( true ); // display frame
   } // end main
} // end class LinesRectsOvals
```

6.6 . Drawing Arcs

An arc is drawn as a portion of an oval. Arc angles are measured in degrees. Arcs sweep (i.e., move along a curve) from a starting angle through the number of degrees specified by their arc angle. The starting angle indicates in degrees where the arc begins. The arc angle specifies the total number of degrees through which the arc sweeps.

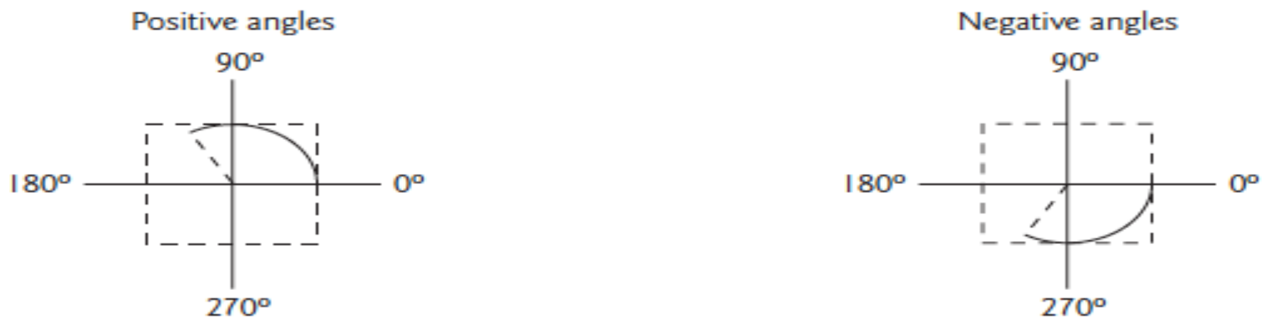


Fig: Positive and Negative arc Angle

The left set of axes shows an arc sweeping from zero degrees to approximately 110 degrees. Arcs that sweep in a *counterclockwise* direction are measured in positive degrees. The set of axes on the right shows an arc sweeping from zero degrees to approximately –110 degrees. Arcs that sweep in a *clockwise* direction are measured in negative degrees. When drawing an arc, we specify a bounding rectangle for an oval. The arc will sweep along part of the oval.

Graphics methods *drawArc* and *fillArc* for drawing arcs are shown below.

Methods	Description
public void drawArc(int x, int y, int width,	Draws an arc relative to the bounding rectangle's top-left

int height, int startAngle, int arcAngle)	x- and y-coordinates with the specified width and height. The arc segment is drawn starting at startAngle and sweeps arcAngle degrees.
public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)	Draws a filled arc (i.e., a sector) relative to the bounding rectangle's top-left x- and y-coordinates with the specified width and height. The arc segment is drawn starting at startAngle and sweeps arcAngle degrees.

Example Program

ArcsJPanel.java

```
// Drawing arcs.
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JPanel;

public class ArcsJPanel extends JPanel
{
   // draw rectangles and arcs
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // call superclass's paintComponent

      // start at 0 and sweep 360 degrees
      g.setColor( Color.RED );
      g.drawRect( 15, 35, 80, 80 );
      g.setColor( Color.BLACK );
      g.drawArc( 15, 35, 80, 80, 0, 360 );

      // start at 0 and sweep 110 degrees
      g.setColor( Color.RED );
      g.drawRect( 100, 35, 80, 80 );
      g.setColor( Color.BLACK );
      g.drawArc( 100, 35, 80, 80, 0, 110 );

      // start at 0 and sweep -270 degrees
      g.setColor( Color.RED );
      g.drawRect( 185, 35, 80, 80 );
      g.setColor( Color.BLACK );
      g.drawArc( 185, 35, 80, 80, 0, -270 );

      // start at 0 and sweep 360 degrees
      g.fillArc( 15, 120, 80, 40, 0, 360 );

      // start at 270 and sweep -90 degrees
      g.fillArc( 100, 120, 80, 40, 270, -90 );

      // start at 0 and sweep -270 degrees
      g.fillArc( 185, 120, 80, 40, 0, -270 );
   } // end method paintComponent
} // end class ArcsJPanel
```

DrawArcs.java

```
// Drawing arcs.
import javax.swing.JFrame;
```

```
public class DrawArcs
{
   // execute application
   public static void main( String[] args )
   {
      // create frame for ArcsJPanel
      JFrame frame = new JFrame( "Drawing Arcs" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      ArcsJPanel arcsJPanel = new ArcsJPanel(); // create ArcsJPanel
      frame.add( arcsJPanel ); // add arcsJPanel to frame
      frame.setSize( 300, 210 ); // set frame size
      frame.setVisible( true ); // display frame
   } // end main
} // end class DrawArcs
```

6.7. Drawing Polygons and Polylines

Polygons are closed multisided shapes composed of straight-line segments. Polylines are sequences of connected points. Some methods require a *Polygon* object (package *java.awt*).

Methods	Description
public void drawPolygon(int[] xPoints, int[] yPoints, int points) .	Draws a polygon. The x-coordinate of each point is specified in the xPoints array and the y-coordinate of each point in the yPoints array. The last argument specifies the number of points. This method draws a closed polygon. If the last point is different from the first, the polygon is closed by a line that connects the last point to the first
public void drawPolyline(int[] xPoints, int[] yPoints, int points)	Draws a sequence of connected lines. The x-coordinate of each point is specified in the xPoints array and the y-coordinate of each point in the yPoints array. The last argument specifies the number of points. If the last point is different from the first, the polyline is not closed.
public void drawPolygon(Polygon p)	Draws the specified polygon.
public void fillPolygon(int[] xPoints, int[] yPoints, int points)	Draws a filled polygon. The x-coordinate of each point is specified in the xPoints array and the y-coordinate of each point in the yPoints array. The last argument specifies the number of points. This method draws a closed polygon. If the last point is different from the first, the polygon is closed by a line that connects the last point to the first.
public void fillPolygon(Polygon p)	Draws the specified filled polygon. The polygon is closed.
public Polygon()	Constructs a new polygon object. The polygon does not contain any points.
public Polygon(int[] xValues, int[] yValues, int numberOfPoints)	Constructs a new polygon object. The polygon has numberOfPoints sides, with each point consisting of an x-coordinate from xValues and a y-coordinate from yValues.
public void addPoint(int x, int y)	Adds pairs of x- and y-coordinates to the Polygon.

Fig: Graphics methods for polygons and class Polygon methods.

PolygonsJPanel.java

```
// Drawing polygons.
import java.awt.Graphics;
import java.awt.Polygon;
import javax.swing.JPanel;

public class PolygonsJPanel extends JPanel
{
   // draw polygons and polylines
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // call superclass's paintComponent

      // draw polygon with Polygon object
      int[] xValues = { 20, 40, 50, 30, 20, 15 };
      int[] yValues = { 50, 50, 60, 80, 80, 60 };
      Polygon polygon1 = new Polygon( xValues, yValues, 6 );
      g.drawPolygon( polygon1 );

      // draw polylines with two arrays
      int[] xValues2 = { 70, 90, 100, 80, 70, 65, 60 };
      int[] yValues2 = { 100, 100, 110, 110, 130, 110, 90 };
      g.drawPolyline( xValues2, yValues2, 7 );

      // fill polygon with two arrays
      int[] xValues3 = { 120, 140, 150, 190 };
      int[] yValues3 = { 40, 70, 80, 60 };
      g.fillPolygon( xValues3, yValues3, 4 );

      // draw filled polygon with Polygon object
      Polygon polygon2 = new Polygon();
      polygon2.addPoint( 165, 135 );
      polygon2.addPoint( 175, 150 );
      polygon2.addPoint( 270, 200 );
      polygon2.addPoint( 200, 220 );
      polygon2.addPoint( 130, 180 );
      g.fillPolygon( polygon2 );
   } // end method paintComponent
} // end class PolygonsJPanel
```

DrawPolygons.java

```
// Drawing polygons.
import javax.swing.JFrame;

public class DrawPolygons
{
   // execute application
   public static void main( String[] args )
   {
      // create frame for PolygonsJPanel
      JFrame frame = new JFrame( "Drawing Polygons" );
      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

      PolygonsJPanel polygonsJPanel = new PolygonsJPanel();
      frame.add( polygonsJPanel ); // add polygonsJPanel to frame
      frame.setSize( 280, 270 ); // set frame size
      frame.setVisible( true ); // display frame
   } // end main
} // end class DrawPolygons
```

More Example Programs

SmileyFace.Java

```
import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class SmilyFace extends JPanel {

      public void paint(Graphics g) {
           g.setColor(Color.yellow);
           g.fillOval(90, 70, 80, 80); // face fill with yellow color

           g.setColor(Color.black);
           g.drawOval(90, 70, 80, 80); // face
           g.drawOval(110, 95, 5, 5); // eye
           g.drawOval(145, 95, 5, 5); // eye
           g.drawLine(130, 95, 130, 115); // nose
           g.drawArc(113, 115, 35, 20, 0, -180); // mouth
      }
      public static void main(String[] args) {
           JFrame frame = new JFrame();
           frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           frame.setSize(1000, 1000);
        frame.setLocationRelativeTo(null);
           frame.getContentPane().add(new SmilyFace());
           frame.setVisible(true);
      }
}

SmileyFaceApplet.java

import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JApplet;

public class SmilyFaceApplet extends JApplet {

      public void paint(Graphics g) {
           g.setColor(Color.yellow);
           g.fillOval(90, 70, 80, 80); // face fill with yellow color
           g.setColor(Color.black);
           g.drawOval(90, 70, 80, 80); // face
           g.drawOval(110, 95, 5, 5); // eye
           g.drawOval(145, 95, 5, 5); // eye
           g.drawLine(130, 95, 130, 115); // nose
           g.drawArc(113, 115, 35, 20, 0, -180); // mouth
      }
}
```

Nepali Flag using Graphics

```
NepalFlag.java
import javax.swing.*;
import java.awt.*;

public class NepalFlag extends JPanel {
    public void paint(Graphics g){

        /* For Left bar of Flag*/
        g.setColor(Color.BLACK);
        g.fillRect(10,20,10,400);

        /*For the outer blue polygon*/
        int x[]={20,200,100,240,20,20};
        int y[]={20,150,150,300,300,20};
        g.setColor(Color.BLUE);
        g.fillPolygon(x,y,6);

        /*For inner red polygon */
        int xx[]={30,170,75,215,30,30};
        int yy[]={40,140,140,290,290,35};
        g.setColor(Color.red);
        g.fillPolygon(xx,yy,6);

        /*for the sun shape */
        g.setColor(Color.WHITE);
        g.fillArc(50,80,40,40,0,-180);

        /*for the moon shape */
        g.fillOval(50,200,50,50);

        Font f1 = new Font("Times New Roman",Font.BOLD,30);
        g.setFont(f1);
        g.setColor(Color.BLACK);
        g.drawString("NEPALI FLAG", 150, 400);
    }
}

NepaliFlagRun.java
import java.awt.*;
import javax.swing.*;
public class NepaliFlagRun {
    public static void main(String [] args)
    {
        JFrame frame = new JFrame("NEPALI FLAG");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        NepalFlag nepal = new NepalFlag();
        frame.add(nepal);
        frame.setSize(500,500);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

Chapter 7: Network Programming

7.1. Introduction

Java's networking support is the concept of a socket. A socket identifies an *endpoint* in a network. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a port, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

The client requests that some action be performed, and the server performs the action and responds to the client. A common implementation of the request-response model is between *web browsers* and *web servers*. When a user selects a website to browse through a browser (the client application), a request is sent to the appropriate web server (the server application). The server normally responds to the client by sending an appropriate web page to be rendered by the browser.

In socket-based communications, which enable applications to view networking as if it were file I/O-a program can read from a socket or write to a socket as simply as reading from a file or writing to a file. The socket is simply a software construct that represents one endpoint of a connection. With stream sockets, a process establishes a connection to another process. While the connection is in place, data flows between the processes in continuous streams. Stream sockets are said to provide a connection-oriented service. The protocol used for transmission is the popular TCP (Transmission Control Protocol). With datagram sockets, individual packets of information are transmitted. The protocol used-UDP, the User Datagram Protocol-is a connectionless service and does not guarantee that packets arrive in any particular order. With UDP, packets can even be lost or duplicated. UDP is most appropriate for network applications that do not require the error checking and reliability of TCP. Stream sockets and the TCP protocol will be more desirable for the vast majority of Java networking applications. Once a connection has been established, a higher-level protocol ensues, which is dependent on which port we are using. TCP/IP reserves the lower *1,024* ports for specific protocols. For Example port number *21 is for FTP*; *23 is for Telnet*; *25 is for e-mail*; *43 is for whois*; *79 is for finger*; *80 is for HTTP*; *119 is for netnews*, and the list goes on. It is up to each protocol to determine how a client should interact with the port.

Java provides a number of built-in networking capabilities that make it easy to develop Internet based and web-based applications. Java can enable programs to search the world for information and to collaborate with programs running on other computers internationally, nationally or just within an organization (subject to security constraints).

Java's fundamental networking capabilities are declared by the classes and interfaces of package java.net, through which Java offers stream-based communications that enable applications to view networking as streams of data. The classes and interfaces of package *java.net* also offer packet-based communications for transmitting individual packets of information-commonly used to transmit data images, audio and video over the Internet.

7.2. Java URL Processing

URL stands for *Uniform Resource Locator* and represents a resource on the World Wide Web, such
as a Web page or FTP directory.
protocol://host:port/path?query#ref

Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority. The following is a URL to a Web page whose protocol is HTTP:
http://www.am rood.com/index.htm?language=en#j2se
Notice that this URL does not specify a port, in which case the default port for the protocol is used. With HTTP, the default port is 80.

7.2.1. URL Class Methods

The java.net.URL class represents a URL and has complete set of methods to manipulate URL in Java. The URL class has several constructors for creating URLs, including the following

SN	Methods with Description
1	public URL(String protocol, String host, int port, String file) throws MalformedURLException Creates a URL by putting together the given parts.
2	public URL(String protocol, String host, String file) throws MalformedURLException Identical to the previous constructor, except that the default port for the given protocol is used.
3	public URL(String url) throws MalformedURLException Creates a URL from the given String
4	public URL(URL context, String url) throws MalformedURLException Creates a URL by parsing the together the URL and String arguments

The URL class contains many methods for accessing the various parts of the URL being represented. Some of the methods in the URL class include the following

SN	Methods and Description
1	**public String getPath():** Returns the path of the URL.
2	**public String getQuery():** Returns the query part of the URL.
3	**public String getAuthority():** Returns the authority of the URL.
4	**public int getPort():** Returns the port of the URL.
5	**public int getDefaultPort():** Returns the default port for the protocol of the URL
6	**public String getProtocol():** Returns the protocol of the URL
7	**public String getHost():** Returns the host of the URL
8	**public String getHost():** Returns the host of the URL
9	**public String getFile():** Returns the filename of the URL
10	**public String getRef():** Returns the reference part of the URL
11	**public URLConnection openConnection() throws IOException** Opens a connection to the URL, allowing a client to communicate with the resource

Example Program

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

```
public class URLDemo
{
public static void main(String[] args) {
try {
URL url = new URL("http://www.amrood.com/index.htm?language=en#j2se");
           System.out.println("URL is " + url.toString());
           System.out.println("protocol is " + url.getProtocol());
           System.out.println("authority is " + url.getAuthority());
           System.out.println("file name is " + url.getFile());
           System.out.println("host is " + url.getHost());
           System.out.println("path is " + url.getPath());
           System.out.println("port is " + url.getPort());
           System.out.println("default port is " + url.getDefaultPort());
           System.out.println("query is " + url.getQuery());
           System.out.println("ref is " + url.getRef());
     } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

7.2.2. URLConnections Class Methods

The openConnection method returns a *java.net.URLConnection*, an abstract class whose subclasses represent the various types of URL connections. For example:
If we connect to a URL whose protocol is HTTP, the *openConnection* method returns an HttpURLConnection object.
If we connect to a URL that represents a JAR file, the *openConnection* method returns a JarURLConnection object.

The *URLConnection* class has many methods for setting or determining information about the connection, including the following:

SN	Methods with Description
1	**Object getContent():** Retrieves the contents of this URL connection.
2	**Object getContent(Class[] classes):** Retrieves the contents of this URL connection
3	**String getContentEncoding():** Returns the value of the content-encoding header field
4	**int getContentLength():** Returns the value of the content-length header field
5	**String getContentType():** Returns the value of the content-type header field
6	**int getLastModified():** Returns the value of the last-modified header field
7	**long getExpiration():** Returns the value of the expires header field
8	**long getIfModifiedSince():** Returns the value of this object's ifModifiedSince field
9	**public void setDoInput(boolean input):** Passes in true to denote that the connection will be used for input. The default value is true because clients typically read from a URLConnection
10	**public void setDoOutput(boolean output):** Passes in true to denote that the connection will be used for output. The default value is false because many types of URLs do not support being written to
11	**public InputStream getInputStream() throws IOException:** Returns the input stream of the URL connection for reading from the resource
12	**public OutputStream getOutputStream() throws IOException:** Returns the output stream of the URL connection for writing to the resource
13	**public URL getURL():** Returns the URL that this URLConnection object is connected to

Example Program

```
import java.net.* ;
import java.io.* ;
public class URLConnDemo
{
public static void main(String[] args) {
        try {
            URL url = new URL("https://www.oracle.com/index.html");
            URLConnection urlConnection = url.openConnection();
            HttpURLConnection connection = null;
            if (urlConnection instanceof HttpURLConnection) {
                connection = (HttpURLConnection) urlConnection;
            } else {
                System.out.println("Please enter an HTTP URL.");
                return;
            }
            BufferedReader in = new BufferedReader(
      new InputStreamReader(connection.getInputStream()));
            String urlString = "";
            String current;
            while ((current = in.readLine()) != null) {
                urlString += current;
            }
            System.out.println(urlString);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

7.2.3. Socket Programming

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a *network connection, a text file, a terminal*, or *something else*. To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as *read()* and *write()* work with sockets in the same way they do with files and pipes. Sockets were first introduced in 2.1BSD and subsequently refined into their current form with 4.2BSD. The sockets feature is now available with most current UNIX system releases.

A Unix Socket is used in a *client-server* application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

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 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 port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and 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, so data can be sent across both streams at the same time.

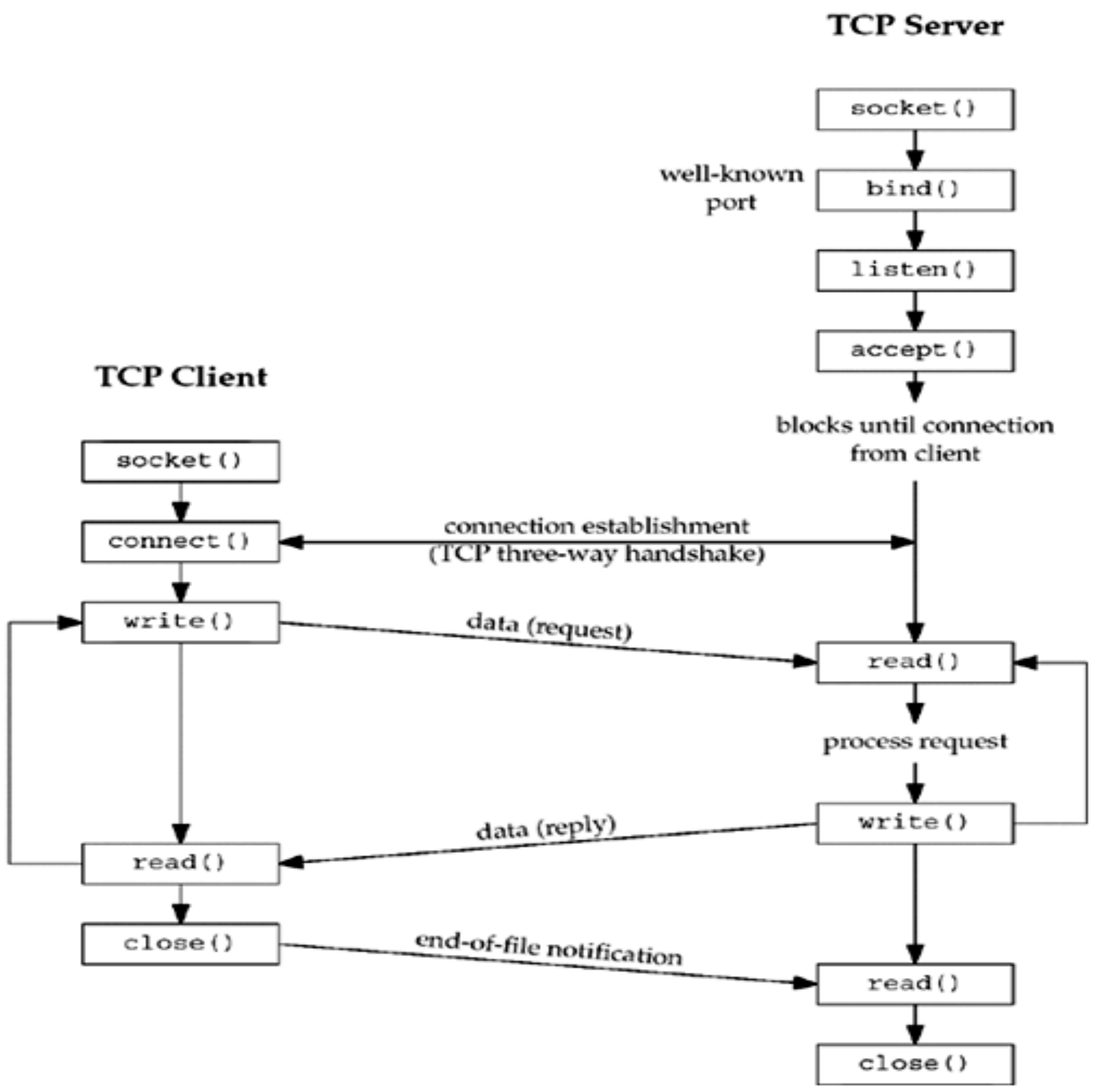


Fig: Client Server Interaction in Unix

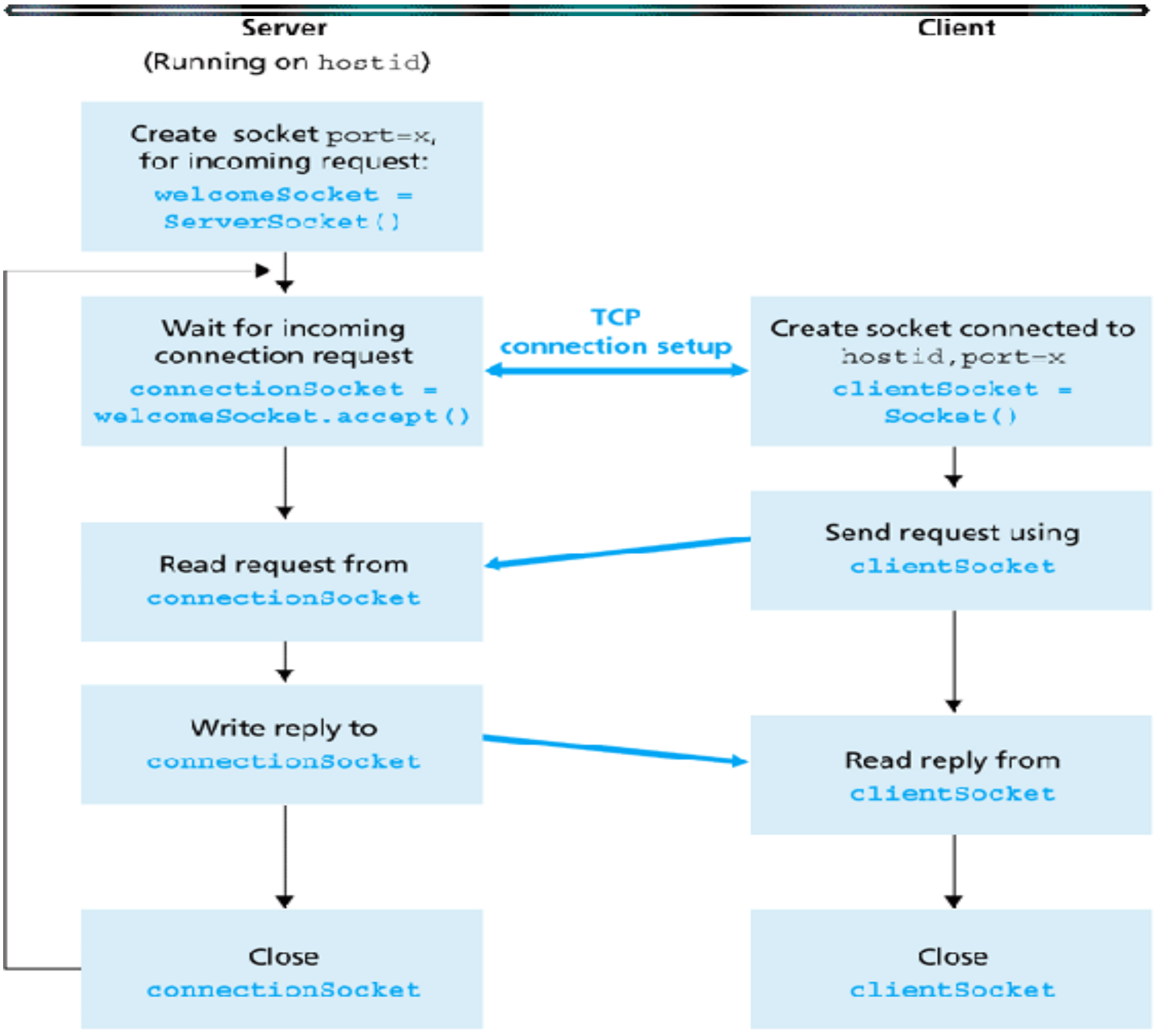

Server
(Running on hostid)
Client
Create socket port=x, for incoming request:
welcomeSocket = ServerSocket()
Wait for incoming connection request
connectionSocket = welcomeSocket.accept()
TCP connection setup
Create socket connected to hostid, port=x
clientSocket = Socket()
Send request using clientSocket
Read request from connectionSocket
Write reply to connectionSocket
Read reply from clientSocket
Close connectionSocket
Close clientSocket

Fig: Client Server Interaction in Java

7.2.3.1. Socket Types

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used. Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

Stream Sockets: Delivery in a networked environment is guaranteed. If we send through the stream socket three items "A, B, C", they will arrive in the same order - "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.

Datagram Sockets: Delivery in a networked environment is not guaranteed. They're connectionless because we don't need to have an open connection as in Stream Sockets - we build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

Raw Sockets: These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

Sequenced Packet Sockets: They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

7.2.3.2. ServerSocket Class Methods

There are following useful classes providing complete set of methods to implement sockets. 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.

SN	Methods with Description
1	**public ServerSocket(int port) throws IOException:** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	**public ServerSocket(int port, int backlog) throws IOException:** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue
3	**public ServerSocket(int port, int backlog, InetAddress address) throws IOException:** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress 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	**public ServerSocket() throws IOException:** Creates an unbound server socket. When using this constructor, use the bind 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.

Common methods of the ServerSocket class

SN	Methods with Description
1	**public int getLocalPort():** Returns the port that the server socket is listening on. This method is useful if we passed in 0 as the port number in a constructor and let the server find a port for us.
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. We can use this method if we 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 server, and communication can begin.

7.2.3.3. Socket Class Methods

The *java.net.Socket* class represents the socket that both the client and 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.

SN	Methods with 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.

SN	Methods with 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 we instantiated 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.

7.2.3.4. InetAddress Class Methods

This class represents an Internet Protocol IP address. Here are following usefull methods which is needed for socket programming.

SN	Methods with Description
1	**static InetAddress getByAddress(byte[] addr):** Returns an InetAddress object given the raw IP address.
2	**static InetAddress getByAddress(String host, byte[]addr):** Create an InetAddress based on the provided host name and IP address
3	**static InetAddress getByName(Stringhost):** 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 getLocalHost():** Returns the local host
7	**String toString():** Converts this IP address to a String

Example Programs

```
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 {
                /*Accept Connection*/
                System.out.println("Waiting for client on port "
                        + serverSocket.getLocalPort() + "...");
                Socket server = serverSocket.accept();
                System.out.println("Just connected to "
                        + server.getRemoteSocketAddress()
                );
                /*Reading Data from Client Socket*/
                DataInputStream in =
                        new DataInputStream(server.getInputStream());
                System.out.println(in.readUTF());
                /*Writing Data to Client Socket*/
                DataOutputStream out =
                        new DataOutputStream(server.getOutputStream());
                out.writeUTF("Thank you for connecting to "
                        + server.getLocalSocketAddress() +
"\nGoodbye!");
                /*Closing Client Socket*/
                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]);
        int port = 60666;
        try {
            Thread t = new GreetingServer(port);
            t.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

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]);
        String serverName = "localhost";
        int port = 60666;

        try {
            /*Creating Socket, Connecting Server*/
            System.out.println("Connecting to " + serverName + " on port
" + port);
        Socket client = new Socket(serverName,port);
        System.out.println("Just connected to " +
client.getRemoteSocketAddress());
        /*Writing Data in Socket*/
        OutputStream outToServer = client.getOutputStream();
            DataOutputStream out = new DataOutputStream(outToServer);
            out.writeUTF("Hello from " +
client.getLocalSocketAddress());
            /*Reading Data from Socket*/
            InputStream inFromServer = client.getInputStream();
            DataInputStream in = new DataInputStream(inFromServer);
            System.out.println("Server says " + in.readUTF());
            client.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}


MyServer.java
import java.io.*;
import java.net.*;

public class MyServer {

    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(6666);
            Socket s = ss.accept();//establishes connection
            DataInputStream dis = new
DataInputStream(s.getInputStream());
            String str = (String) dis.readUTF();
            System.out.println("message= " + str);
            ss.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

MyClient.java
```

```
import java.io.*;
import java.net.*;
public class MyClient {
public static void main(String[] args) {
try{
Socket s=new Socket("localhost",6666);
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
dout.writeUTF("Hello Server");
dout.flush();
dout.close();
s.close();
}catch(Exception e){
System.out.println(e);
}
}
}
```

7.3. Multicast Programming with Java

Point-to-point communication is often referred as *unicast*. Although point-to-point communications are good for many applications, there are situations in which we need a different model which need to communicate in one-to-many model. *Broadcast* (i.e. one-to-all communication) may serve for this purpose, its overhead is too large that make it an impractical solution.

Another approach is to use many **unicasts** for the one-to-many model. *Multicast is the delivery of information to multiple destinations simultaneously using the most efficient strategy to deliver the messages over the network over each link of the network only once and only create copies when the links to the destinations split.*

Multicast is a special feature of UDP protocol that enable programmer to send message to a group of receivers on a specific multicast IP address and port. Multicast has advantage in this scenario.

7.3.1. Multicast Characteristics

- Multicast is using UDP under the hood. So sending and receiving data are much the same as UDP
- The big noticeable from UDP Sender should address packages to an IP number in the range between **224.0.0.1**and **239.255.255.254**.
- Receivers must join multicast group to receive packet
- Several multicast sockets can be bound simultaneously to the same port (Contrary to UDP and TCP)
- Multicast is viable for video conference, service discovery application, etc.

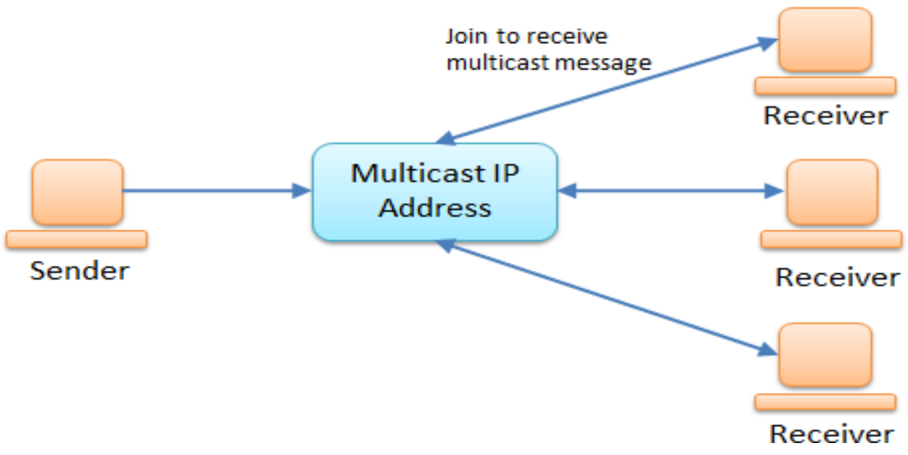


Fig: Multicast Network

Example

File: MulticasSender.java

```
import java.io.*;
import java.net.*;
public class MulticastSender {

    public static void main(String[] args) {
        DatagramSocket socket = null;
        DatagramPacket outPacket = null;
        byte[] outBuf;
        final int PORT = 8888;

        try {
            socket = new DatagramSocket();
            long counter = 0;
            String msg;
            while (true) {
                msg = "This is multicast! " + counter;
                counter++;
                outBuf = msg.getBytes();
                //Send to multicast IP address and port
              InetAddress address = InetAddress.getByName("224.2.2.3");
       outPacket=new DatagramPacket(outBuf,outBuf.length,address,PORT);

                socket.send(outPacket);
                System.out.println("Server sends : " + msg);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException ie) {
                } //End of while
            } //End of outer try
        } catch (IOException ioe) {
            System.out.println(ioe);
        } //End of catch
    }//End of main
} //End of class
```

File: MulticastReceiver.java
```
import java.io.*;
import java.net.*;

public class MulticastReceiver {

    public static void main(String[] args) {
        MulticastSocket socket = null;
        DatagramPacket inPacket = null;
        byte[] inBuf = new byte[256];
        try {
            //Prepare to join multicast group
            socket = new MulticastSocket(8888);
            InetAddress address = InetAddress.getByName("224.2.2.3");
            socket.joinGroup(address);

            while (true) {
                inPacket = new DatagramPacket(inBuf, inBuf.length);
                socket.receive(inPacket);
                String msg = new String(inBuf, 0, inPacket.getLength());
         System.out.println("From"+inPacket.getAddress()+"Msg:" + msg);
        }
               } catch (IOException ioe) {
             System.out.println(ioe);
     }
} //End of main
} //End of class
```

Chapter 8: Java Database Connectivity (JDBC)

8.1. Introduction

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. Some popular relational database management systems (RDBMSs) are Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL and MySQL. The JDK now comes with a pure-Java RDBMS called Java DB-Oracles's version of Apache Derby.

Java programs communicate with databases and manipulate their data using the *Java Database Connectivity (JDBC)* API. A *JDBC* driver enables Java applications to connect to a database in a particular DBMS and allows us to manipulate that database using the *JDBC API*. Most popular database management systems now provide *JDBC drivers*. There are also many third-party JDBC drivers available.

The JDBC consists of two layers. The top layer is the *JDBC API*. This API communicates with the *JDBC manager driver API*, sending it the various SQL statements. The manager should communicate with the various third-party drivers that actually connect to the database and return the information from the query or perform the action specified by the query.

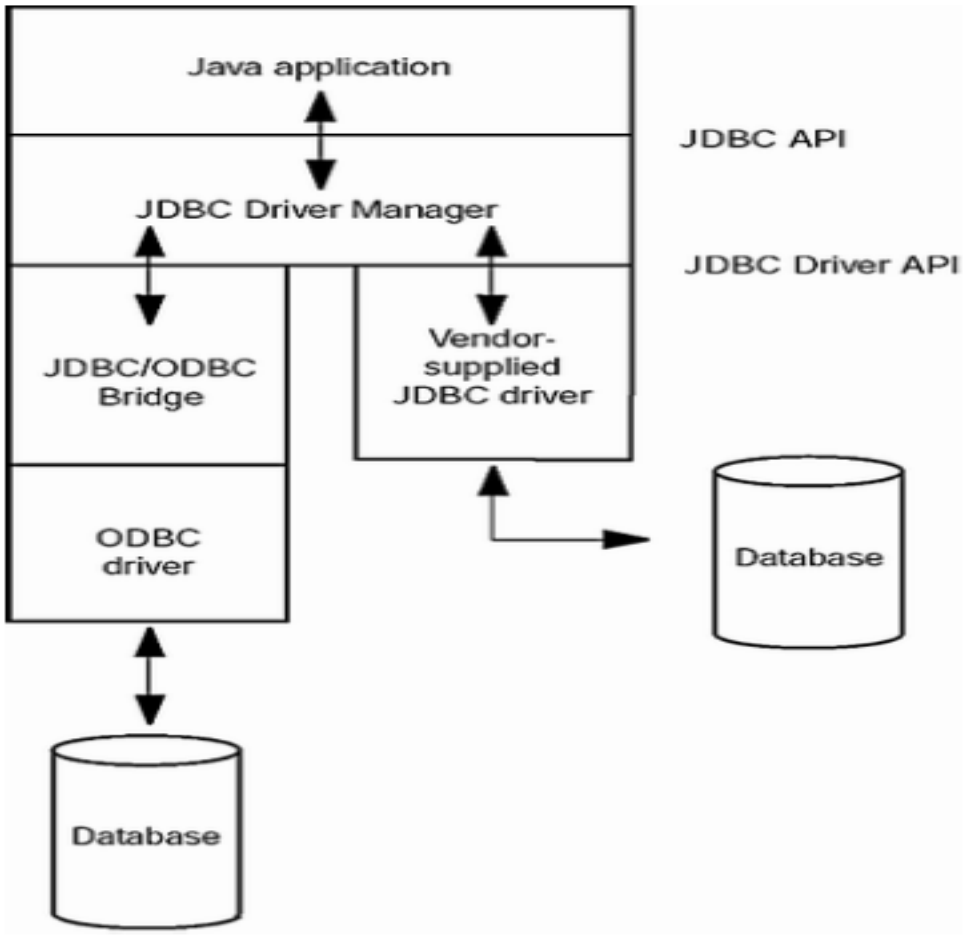


Fig: JDBC-to-database communication path

8.2. JDBC Driver

JDBC is: Java Database Connectivity

- is a Java API for connecting programs written in Java to the data in relational databases.
- consists of a set of classes and interfaces written in the Java programming language.
- provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.
- The standard defined by Sun Microsystems, allowing individual providers to implement and extend the standard with their own JDBC drivers.
- The ***Java.sql*** package 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:

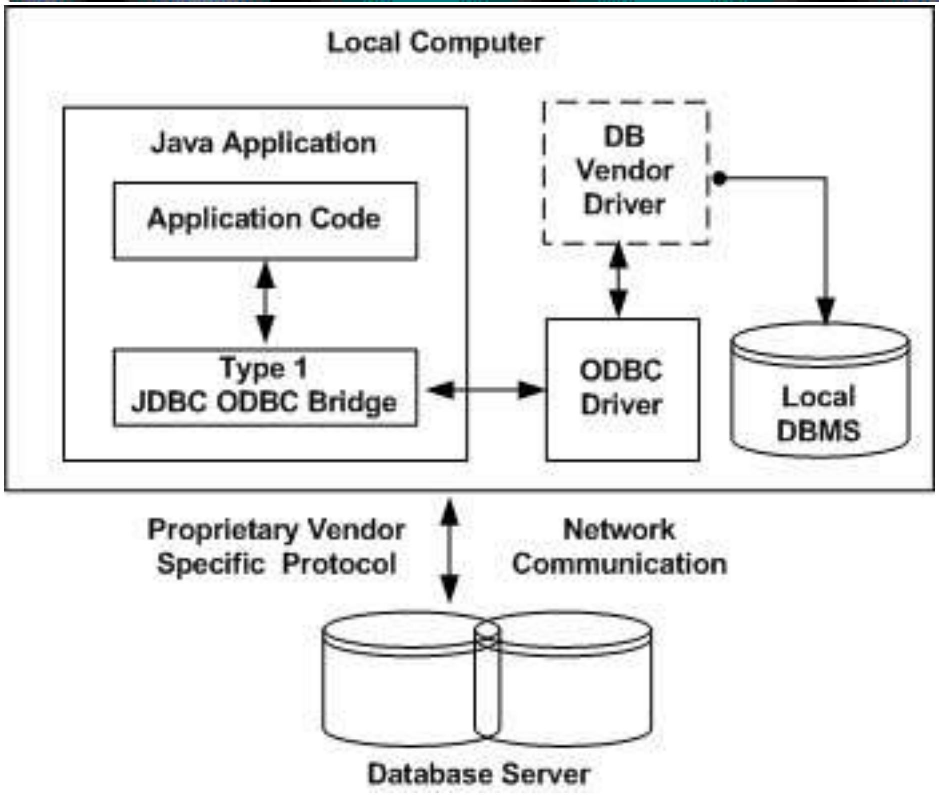
- establishes a connection with a database
- sends SQL statements
- processes the results.

8.2.1. 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.

8.2.1.1. Type 1: JDBC-ODBC Bridge Driver

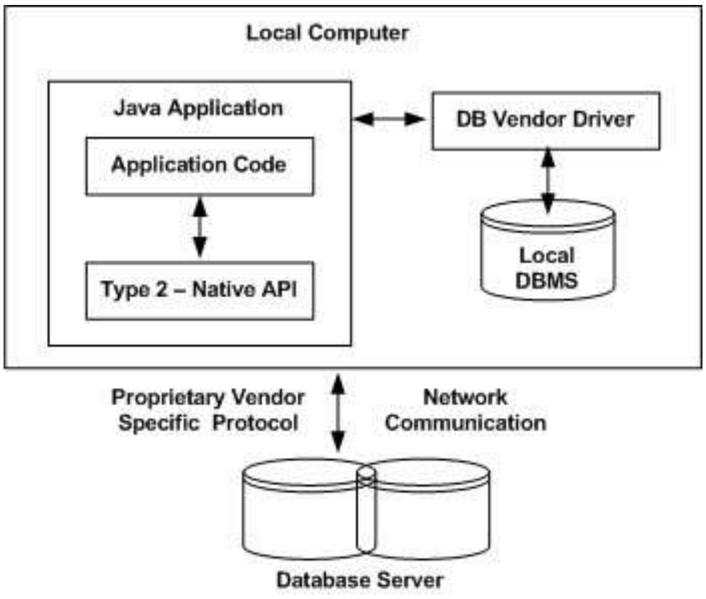
In a **Type 1 driver**, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on our system a Data Source Name (DSN) that represents the target database. When Java first came out, this was a useful driver because most databases only supported 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.

8.2.1.2. Type 2: JDBC-Native API

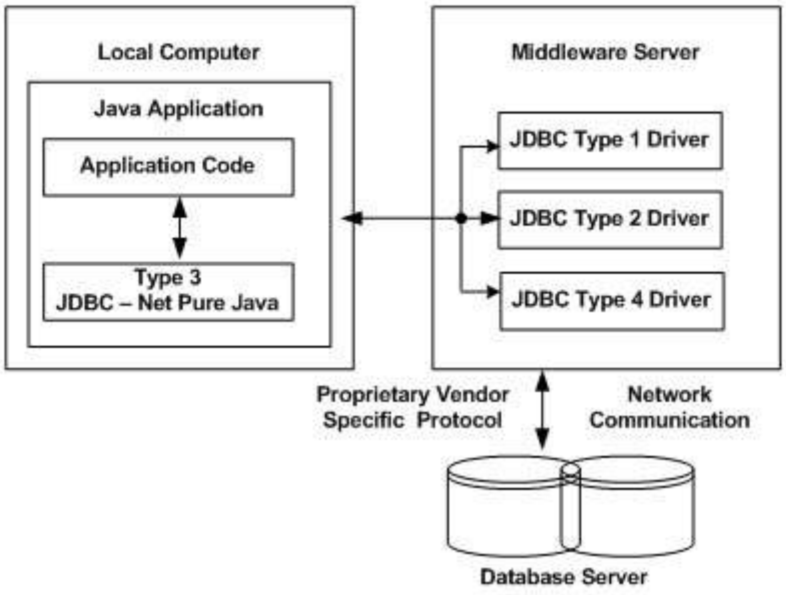
In a **Type 2 driver**, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine. If 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 we 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.

8.2.1.3. Type 3: JDBC-Net pure Java

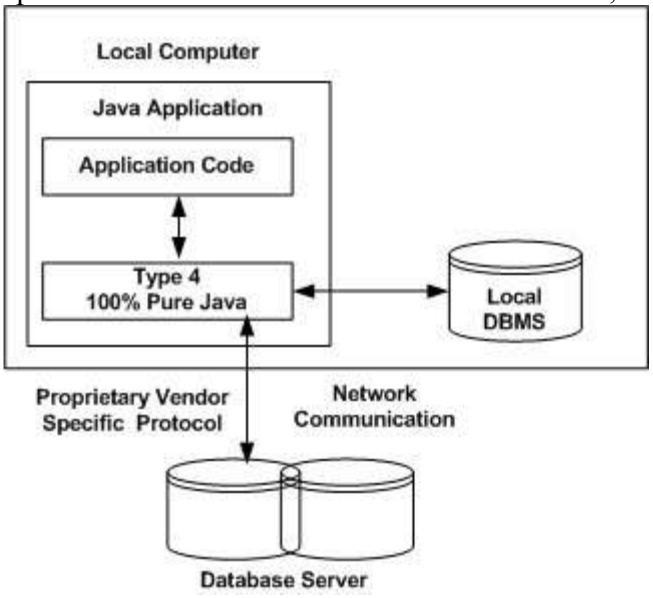
In a **Type 3 driver**, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded 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.



We can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, we need some knowledge of the application server's configuration in order to effectively use this driver type.

8.2.1.4. 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; we 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.

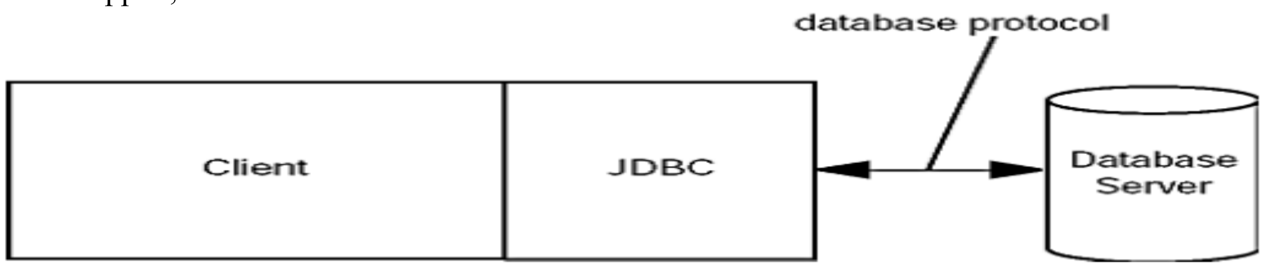
Which Driver should be Used?

- If we are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If our 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 our database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

8.3. Typical Uses of JDBC

We can use JDBC in both applications and applets. In an applet, all the normal security restrictions apply. By default, the security manager assumes that all applets written in the Java programming language are untrusted.
In particular, applets that use JDBC are only able to open a database connection to the server from which they are downloaded. That means the Web server and the database server must be the same machine, which is not a typical setup. Of course, the Web server can have a proxy service that routes database traffic to another machine. With signed applets, this restriction can be loosened.

Applications, on the other hand, have complete freedom to access remote database servers. If we implement a traditional client/server program, it probably makes more sense to use an application, not an applet, for database access.



However, the world is moving away from client/server and toward a "three-tier model" or even more advanced "n-tier models." In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates **visual presentation** (on the client) from the **business logic** (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when we use a web browser as the client), RMI (when we use an application or applet), or another mechanism. *JDBC is used to manage the communication between the middle tier and the back-end database*. Following figure shows the basic architecture.

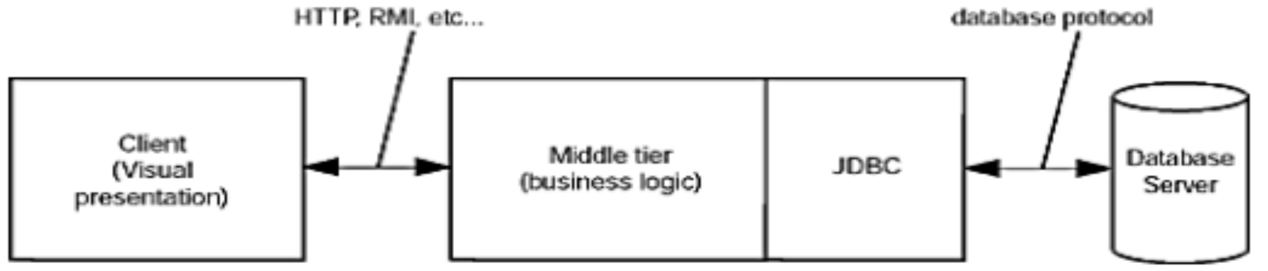


Fig: 3-tier Application

SQL is the industry-standard approach to accessing relational databases. JDBC supports SQL, enabling developers to use a wide range of database formats without knowing the specifics of the underlying database. JDBC also supports the use of database queries specific to a database format.

The JDBC class library's approach to accessing databases with SQL is comparable to existing database-development techniques, so interacting with an SQL database by using JDBC isn't much different than using traditional database tools. Java programmers who already have some database experience can hit the ground running with JDBC. The JDBC library includes classes for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating a statement using SQL
- Executing that SQL query in the database
- Viewing the resulting records

These JDBC classes are all part of the *java.sql* package.

8.4. Database Drivers

Java programs that use JDBC classes can follow the familiar programming model of issuing SQL statements and processing the resulting data. The format of the database and the platform it was prepared on don't matter.

A driver manager makes the platform and database independence possible. The classes of the *JDBC class library are largely dependent on driver managers*, which keep track of the drivers required to access database records. We'll need a different driver for each database format that's used in a program, and sometimes we might need several drivers for versions of the same format. *Java DB includes its own driver. JDBC also includes a driver that bridges JDBC and another database-connectivity standard, ODBC.*

The JDBC-ODBC bridge allows JDBC drivers to be used as ODBC drivers by converting JDBC method calls into ODBC function calls.
Using the JDBC-ODBC bridge requires three things:

- The JDBC-ODBC bridge driver included with Java: *sun.jdbc.odbc.JdbcOdbcDriver*
- An ODBC driver
- An ODBC data source that has been associated with the driver using software such as the ODBC Data Source Administrator.

8.5. Manipulating Databases with JDBC

Steps Required to access Database using JDBC

There are following steps required to access Database using JDBC application:

1. **Import the packages:** Requires that we need to include the packages containing the JDBC classes needed for database programming. Most often, using ***import java.sql.**** will suffice.
2. **Register the JDBC driver:** Requires that we initialize a driver so we can open a communications channel with the database.
3. **Open a connection:** Requires using the ***DriverManager.getConnection()*** method to create a Connection object, which represents a physical connection with a **selected** database.
4. **Creating a Statement Object:** Create the required queries.
5. **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to perform operations in a table.
6. **Processing the Result Set**
7. **Closing the Result Set and Statement Objects**
8. **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

8.5.1. Connecting to and Querying a Database

The following section of program performs a simple query on the *books* database that retrieves the entire *Authors* table and displays the data. The program illustrates *connecting to the database*, *querying the database and processing the result*. The discussion that follows presents the key JDBC aspects of the program.

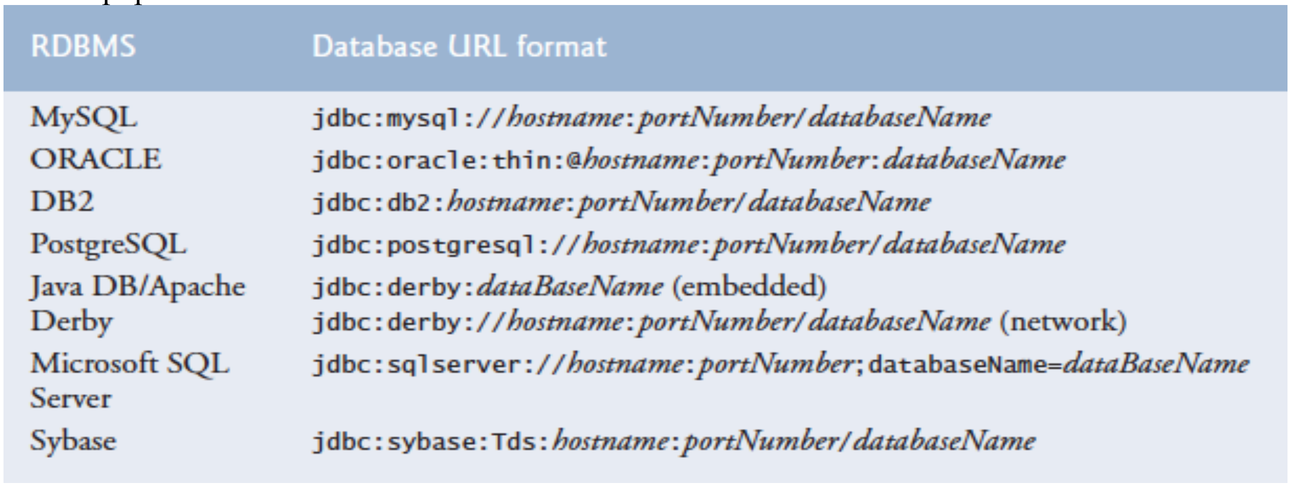
DisplayAuthors.java

```
// Displaying the contents of the authors table.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DisplayAuthors
{
   // database URL
   static final String DATABASE_URL =
"jdbc:mysql://localhost:3306/books";

   // launch the application
   public static void main( String args[] )
   {
      Connection connection = null; // manages connection
      Statement statement = null; // query statement
      ResultSet resultSet = null; // manages results

      // connect to database books and query database
      try
      {
         // establish connection to database
         connection = DriverManager.getConnection(
            DATABASE_URL, "root", "" );

         // create Statement for querying database
         statement = connection.createStatement();

         // query database
         resultSet = statement.executeQuery(
            "SELECT authorID, firstName, lastName FROM authors");

         // process query results
         ResultSetMetaData metaData = resultSet.getMetaData();
         int numberOfColumns = metaData.getColumnCount();
         System.out.println( "Authors Table of Books Database:\n" );

         for ( int i = 1; i <= numberOfColumns; i++ )
            System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
         System.out.println();

         while ( resultSet.next() )
         {
            for ( int i = 1; i <= numberOfColumns; i++ )
               System.out.printf( "%-8s\t", resultSet.getObject( i ) );
            System.out.println();
         } // end while
      }  // end try
      catch ( SQLException sqlException )
      {
         sqlException.printStackTrace();
      } // end catch
      finally // ensure resultSet, statement and connection are closed
      {
         try
         {
            resultSet.close();
            statement.close();
            connection.close();
         } // end try
         catch ( Exception exception )
         {
            exception.printStackTrace();
         } // end catch
      } // end finally
   } // end main
} // end class DisplayAuthors
```

In this program there is an import the JDBC interfaces and classes from package ***java.sql*** is used and the declaration of a string constant for the *database URL*. This identifies the name of the database to connect to, as well as information about the *protocol* used by the JDBC driver. Method main connects to the *books* database, queries the database, displays the result of the query and closes the database connection.

8.5.2. Connecting to the Database

In an above program *connection=DriverManager.getConnection(DATABASE_URL,"root","");* create a *Connection* object (package *java.sql*) referenced by *connection*. An object that implements interface ***Connection*** manages the connection between the *Java program and the database*. Connection objects enable programs to create SQL statements that manipulate databases. The program initializes connection with the result of a call to static method ***getConnection*** of class ***DriverManager*** (package ***java.sql***), which attempts to connect to the database specified by its URL. Method *getConnection*() takes three arguments-a String that specifies the *database URL*, a String that specifies the *username* and a String that specifies the *password*. The URL locates the database (possibly on a network or in the local file system of the computer). The URL ***jdbc:mysql://localhost:3306/books*** specifies the protocol for communication (**jdbc**), the sub protocol for communication (***mysql***) and the location of the database ***(//localhost:3306/books***, where *localhost* is the host running the MySQL server , *3306* is default *mysql* running port and ***books*** is the database name). The sub protocol *mysql* indicates that the program uses a MySQL-specific sub protocol to connect to the MySQL database. If the *DriverManager* cannot connect to the database, method *getConnection* throws a *SQLException* (package *java.sql*). Following Figure lists the JDBC driver names and database URL formats of several popular RDBMSs.



RDBMS	Database URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
ORACLE	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`:`*databaseName*
DB2	`jdbc:db2:`*hostname*`:`*portNumber*`/`*databaseName*
PostgreSQL	`jdbc:postgresql://`*hostname*`:`*portNumber*`/`*databaseName*
Java DB/Apache Derby	`jdbc:derby:`*dataBaseName* (embedded) `jdbc:derby://`*hostname*`:`*portNumber*`/`*databaseName* (network)
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

Fig: Popular JDBC database URL formats

8.5.3.Creating a *Statement* for Executing Queries

A *Connection* method ***createStatement*** to obtain an object that implements interface ***Statement*** (package ***java.sql***). The program uses the ***Statement*** object to submit SQL statements to the database.

Type of Statements

There are 3 different kinds of statements.

I) Statement: Used to implement simple SQL statements with no parameters.

Eg: *statement = connection.createStatement();*

II) PreparedStatement: (Extends *Statement*.) Used for precompiling SQL statements that might contain input parameters. Eg:

```
PreparedStatement pstmt = null;
String SQL = " SELECT authorID, firstName, lastName FROM authors ";
pstmt = conn.prepareStatement(SQL);
```

III) CallableStatement: (Extends *PreparedStatement*.) Used to execute stored procedures that may contain both input and output parameters.

Suppose we have a store procedure *getAuthor* in MySql to retrieve authors.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS getAuthor $$
CREATE PROCEDURE getAuthor
  (OUT authorID int, OUT FirstName VARCHAR(255), OUT LastName VARCHAR(255))
BEGIN
  SELECT authorID, firstName, lastName FROM authors;
END $$
```

In order to call those store procedure in program.

```
CallableStatement cs = null;
cs = this.conn.prepareCall("{call getAuthor()}");
ResultSet rs = cs.executeQuery();
```

Three types of parameters exist: IN, OUT, and INOUT

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. We bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. We retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. We bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

8.5.4. Executing a Query

In above program use the *Statement* object's ***executeQuery*** method to submit a query that selects all the author information from table ***Authors***. This method returns an object that implements interface ***ResultSet*** and contains the query results. The ***ResultSet*** methods enable the program to manipulate the query result.

- execute: Returns true if the first object that the query returns is a ***ResultSet*** object. Use this method if the query could return one or more *ResultSet* objects. Retrieve the *ResultSet* objects returned from the query by repeatedly calling *Statement.getResultSet*.
- executeQuery: Returns one ResultSet object.
- executeUpdate: Returns an integer representing the number of rows affected by the SQL statement. Use this method if we are using INSERT,DELETE, or UPDATE SQL statements.

8.5.5. Processing a Query's *ResultSet*

An example of processing of Query ***ResultSet*** is shown in above. First of all it needs to obtains the metadata for the ***ResultSet*** as a ***ResultSetMetaData*** (package ***java.sql***) object. The ***metadata*** describes the ***ResultSet***'s contents. Programs can use metadata programmatically to obtain information about the ***ResultSet***'s column names and types. ResultSetMetaData method ***getColumnCount*** are use to retrieve the number of columns in the ***ResultSet***. First *for loop* is used to display the column names while second *for loop* display the data in each *ResultSet* row. First, the program positions the *ResultSet* cursor (which points to the row being processed) to the first row in the *ResultSet* with method *next*. Method *next* returns *boolean* value *true* if it's able to position to the next row; otherwise, the method returns *false*.

If there are rows in the *ResultSet*, second for loop extract and display the contents of each column in the current row. When a *ResultSet* is processed, each column can be extracted as a specific Java type. In fact, *ResultSetMetaData* method ***getColumnType*** returns a constant integer from class **Types** (package *java.sql*) indicating the type of a specified column. Programs can use these values in a *switch* statement to invoke *ResultSet* methods that return the column values as appropriate Java types. If the type of a column is *Types.INTEGER*, *ResultSet* method *getInt* returns the column value as an *int*. *ResultSet* get methods typically receive as an argument either a column number (as an *int*) or a column name (as a *String*) indicating which column's value to obtain.

For simplicity, this example treats each value as an *Object*. We retrieve each column value with *ResultSet* method ***getObject*** then print the *Object's String* representation. Unlike array indices, *ResultSet* column numbers start at 1. The *finally* block closes the *ResultSet*, the *Statement* and the database *Connection*. *NullPointerException* will throw if the *ResultSet*, *Statement* or *Connection* objects were not created properly.

We can also access the data in a ***ResultSet*** object through a cursor. This cursor is not a database cursor. This cursor is a pointer that points to one row of data in the *ResultSet* object. Initially, the cursor is positioned before the first row. We call various methods defined in the *ResultSet* object to move the cursor.

```
try {
stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
int ID = resultSet.getInt("AuthorID");
          String Fname = resultSet.getString("FirstName");
          String Lname = resultSet.getString("LastName");
            //System.out.println();
             System.out.println(ID +"\t\t" + Fname +"\t\t"+ Lname);
         } // end while
      }  // end try
      catch ( SQLException sqlException )
      {
         sqlException.printStackTrace();
      }
```

Another Example:

//STEP 1. Import required packages

```
import java.sql.*;
public class JDBCExample {
      // JDBC driver name and database URL
      static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
      static final String DB_URL = "jdbc:mysql://localhost/STUDENTS";
      // Database credentials
      static final String USER = "username";
      static final String PASS = "password";
      public static void main(String[] args) {
            Connection conn = null;
            Statement stmt = null;
            try{
                  //STEP 2: Register JDBC driver
                  Class.forName("com.mysql.jdbc.Driver");

                  //STEP 3: Open a connection
                  System.out.println("Connecting to a selected database...");
                  conn = DriverManager.getConnection(DB_URL, USER, PASS);
                  System.out.println("Connected database successfully...");

                  System.out.println("Creating table in given database...");

                  //STEP 4: Creating the statement
                  stmt = conn.createStatement();
                  String sql = "CREATE TABLE REGISTRATION " +
                              "(id INTEGER not NULL, " +
                              " first VARCHAR(255), " +
                              " last VARCHAR(255), " +
                              " age INTEGER, " +
                              " PRIMARY KEY ( id ))";

                  //STEP 5: Execute a query
                  stmt.executeUpdate(sql);
                  System.out.println("Created table in given database...");

                  System.out.println("Inserting records into the table...");
                  stmt = conn.createStatement();
      String sql = "INSERT INTO Registration VALUES (100, 'Zara', 'Ali', 18)";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO Registration VALUES (101, 'Mahnaz', 'Fatma', 25)";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO Registration VALUES (102, 'Zaid', 'Khan', 30)";
      stmt.executeUpdate(sql);

      sql = "INSERT INTO Registration VALUES(103, 'Sumit', 'Mittal', 28)";
      stmt.executeUpdate(sql);

      System.out.println("Inserted records into the table...");
                  }catch(SQLException se){
                        //Handle errors for JDBC
                        se.printStackTrace();
                  }catch(Exception e){
                        //Handle errors for Class.forName
                        e.printStackTrace();
                  }finally{
                        //Step6: Clean up the environment
                        try{
                              if(stmt!=null)
                                    stmt.close();
                        }catch(SQLException se){
                  }// do nothing
                  try{
                        if(conn!=null)
                              conn.close();
                        }catch(SQLException se){
                              se.printStackTrace();
                        }//end finally try
                  }//end try
            System.out.println("Goodbye!");
      }//end main
}//end JDBCExample
```

8.6. ODBC (Open Database Connectivity)

ODBC is

- A standard or open application programming interface (API) for accessing a database.
- SQL Access Group, chiefly Microsoft, in 1992
- By using ODBC statements in a program, we can access files in a number of different databases, including **Access, dBase, DB2, Excel**, and **Text**.
- It allows programs to use SQL requests that will access databases without having to know the proprietary interfaces to the databases.
- ODBC handles the SQL request and converts it into a request the individual database system understands.
- we need the ODBC software, and a separate module or driver for each database to be accessed. Library that is dynamically connected to the application.
- Driver masks the heterogeneity of DBMS operating system and network protocol.
- E.g. (Sybase, Windows/NT, Novell driver)

8.6.1.JDBC vs ODBC

JDBC	ODBC
JDBC is object oriented.	ODBC is procedural oriented.
JDBC is used to provide connection between JAVA and database(oracle,sybase,DB2,ms-access).	ODBC is used to provide connection between front-end application (other than java) and back-end (database like ms-access)
JDBC is for java applications.	ODBC is for Microsoft
JDBC is used by Java programmers to connect to databases	ODBC is used between applications of different types.
JDBC allows SQL-based database access for EJB persistence and for direct manipulation from CORBA, DJB or other server objects	With a small "bridge" program, we can use the JDBC interface to access ODBC accessible databases.
JDBC is multi-threaded	ODBC is not multi-threaded
We can do everything with JDBC that we can do with ODBC, on any platform.	ODBC can't be directly used with Java because it uses a C interface.
	ODBC makes use of pointers which have been removed totally from java.
JDBC is faster	ODBC is slower
JDBC API is a natural Java Interface and is built on ODBC, and therefore JDBC retains some of the basic feature of ODBC	

References

Herbert Schildt; Java: The Complete Reference; Seventh Edition;Tata McGraw-Hill Education
Paul Deitel, Harvey Deitel;Java How To Program;Nineth Edition;Prentice Hall
Java Tutorial;Web-www.tutorialpoints.com