unit 1.pdf

Unit2and3_user interface with swing.pdf

Unit4Database Connectivity.pdf

Unit5Network Programming.pdf

Unit6Java Beans.pdf

Unit7Servlets and JSP.pdf

Unit8RMI and CORBA.pdf

Class

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

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

The general form of class

```
class classname {
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;
        type methodname1(parameter-list) {
        // body of method1
        }
        type methodname2(parameter-list) {
        // body of method2
        }
        // ...
        type methodnameN(parameter-list) {
        // body of methodN
        }
                }
```

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**.

Example of Class

```
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 BoxDemo {
        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);
        }
}
```

Creating Objects

When you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

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

Constructors

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.

/* Here, Box uses a constructor to initialize the dimensions of a box.*/

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

        // This is the constructor for Box.
        Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
        }

        // compute and return volume
        double volume() {
        return width * height * depth;
        }
        }
        class BoxDemo6 {
        public static void main(String args[]) {

        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

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

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

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

```
Constructing Box
Constructing Box
```

Volume is 1000.0
Volume is 1000.0

Overloading Methods

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

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

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

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

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

        // overload test for a double parameter
        double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
}
}
class Overload {
public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // call all versions of test()
```

```
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Overloading Constructors

In addition to overloading normal methods, one can also overload constructors.

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

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

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

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

        // compute and return volume
        double volume() {
```

```
        return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {

        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;

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

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

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

The output produced by this program is shown here:
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0

Static Variables and Methods

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the **keyword static**. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static.

The most common example of a **static member** is **main( ).** main( ) is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as **static** have several restrictions:
■ They can only call other static methods.
■ They must only access static data.
■ They cannot refer to this or super in any way. (The keyword super relates to inheritance.)
If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.

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

Here is the output of the program:
Static block initialized.
x = 42
a = 3
b = 12

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

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.
classname.method( )

Example
Inside main( ), the static method callme( ) and the static variable b are accessed outside of their class.

```
class StaticDemo {
        static int a = 42;
        static int b = 99;
        static void callme() {
        System.out.println("a = " + a);
        }
}
class StaticByName {
        public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
}
}
```

Here is the output of this program:
a = 42
b = 99

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

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

Arrays
In Java Arrays are implemented as objects.

```
// This program demonstrates the length array member.

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

```
}
}
```
This program displays the following output:
length of a1 is 10
length of a2 is 8
length of a3 is 4

Inheritance

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

To inherit a class, you simply incorporate the definition of one class into another by using the **extends keyword**.

```
// A simple example of inheritance.

// Create a superclass.
class A {
        int i, j;
        void showij( ) {
        System.out.println("i and j: " + i + " " + j);
}
}

// Create a subclass by extending class A.
class B extends A {
        int k;
        void showk( ) {
        System.out.println("k: " + k);
        }
        void sum( ) {
        System.out.println("i+j+k: " + (i+j+k));
        }
        }

class SimpleInheritance {
        public static void main(String args[ ]) {
        A superOb = new A( );
```

```
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        // The subclass has access to all public members of its superclass.
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij( );
        subOb.showk( );
        System.out.println( );
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum( );
        }
}
```

The output from this program is shown here:
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24

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

Note: 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.**

Super Keyword

super has two general forms.

- The first calls the superclass' constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

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

super(parameter-list);

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

Example

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

        // construct clone of an object
        Box(Box ob) {                   // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
        }
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
        }

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

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

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

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
```

```
        double weight; // weight of box

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

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

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

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

```
        System.out.println();
        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
        System.out.println("Weight of myclone is " + myclone.weight);
        System.out.println();
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
        System.out.println("Weight of mycube is " + mycube.weight);
        System.out.println();
        }
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

A Second Use for super

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.

This usage has the following general form:

super.member

Here, member can be either a method or an instance variable.

Consider this simple

```
// Using super to overcome name hiding.
class A {
int i;
int j;
}

// 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,int c) {
        super.i = a; // i in A
        i = b; // i in B
```

```
        j=c;
        }

        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, 2,3);
        subOb.show();
        }
}
```

This program displays the following:
i in superclass: 1
i in subclass: 2

Package

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

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

Defining a package

package pkg_name;

Here,package is a keyword and pkg_name is the name of the package. For example, the following statement creates a package called MyPackage.

package MyPackage;

In Java a hierarchy of packages can be created. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];

A package hierarchy must be reflected in the file system of Java development system. For example, a package declared as

package java.awt.image;
needs to be stored in java\awt\image on the Windows file system.

Access Modifiers

The three access specifiers, **private, public, and protected**, provide a variety of ways to produce the many levels of access required by these categories. Table below sums up the interactions

	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 9-1. *Class Member Access*

An Access Example

```
//This is file Protection.java:
package p1;
public class Protection {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;

        public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}

//This is file Derived.java:

package p1;
class Derived extends Protection {
        Derived() {
        System.out.println("derived constructor");
```

```
        System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}

//This is file SamePackage.java:
package p1;
class SamePackage {
        SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
        }
}

//This is file Protection2.java:
package p2;
class Protection2 extends p1.Protection {
        Protection2() {
        System.out.println("derived other package constructor");
        // class or package only
        // System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}

//This is file OtherPackage.java:
package p2;
class OtherPackage {
        OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");
        // class or package only
        // System.out.println("n = " + p.n);
        // class only
```

```
        // System.out.println("n_pri = " + p.n_pri);
        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
}
}

// This is file Demo.java in  package p1.
package p1;
public class Demo {
        public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
        }
}

//This is file Demo.java in package p2.
package p2;
public class Demo {
        public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
        }
}
```

Importing Packages

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

The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing.

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement:

```
import pkg1[.pkg2].(classname|*);
```

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify

either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

import java.util.Date;
import java.io.*;

Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the import statement looks like this:

```
class MyDate extends java.util.Date {
}
```

NOTE: When a package is imported, only those items within the package declared as **public** will be available to **non-subclasses** in the importing code.

For example, if you want the Balance class of the package MyPack to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

```
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);
        }
```

```
}
```

As you can see, 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, you may use Balance
        class and call its constructor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);
        test.show(); // you may also call show()
        }
        }
```

Experiment- Remove the public specifier from the Balance class and then try compiling TestBalance. As explained, errors will result.

Interfaces

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

Using interface, you can specify what a class must do, but not how it does it.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the **"one interface, multiple methods"** aspect of polymorphism. Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.

Defining an Interface

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

***access* interface *name* {**

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

Here, **access** is either **public or not used**. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. **name** is the name of the interface, and can be any valid identifier.

Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. **They must also be initialized with a constant value**. All methods and variables are implicitly public if the interface, itself, is declared as public.

Implementing Interface

```
interface Callback {
        void callback(int param);
}
--Save this file as Callback.java

class Client implements Callback {
        // Implement Callback's interface

        public void callback(int p) {
        System.out.println("callback called with " + p);
        }

        void nonIfaceMeth() {
        System.out.println("Classes that implement interfaces " +
        "may also define other members, too.");
        }
}

// Another implementation of Callback.
```

```
class AnotherClient implements Callback {
        // Implement Callback's interface

        public void callback(int p) {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
}
}

class TestIface {
        public static void main(String args[ ]) {
        Client ob1 = new Client();
        AnotherClient ob2 = new AnotherClient();
        ob1.callback(42);
        ob2.callback(42);
        }
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

Exception-Handling

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. This approach is as cumbersome as it is troublesome.Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Java exception handling is managed via **five keywords**: **try, catch, throw, throws, and finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the try block, it is thrown. Your 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 before a method returns is put in a **finally** block.

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 before try block ends
}
```

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

Uncaught Exceptions

```
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 exception is caught by the default handler provided by the Java run-time system.

Here is the output generated when this example is executed.
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)

Using try and catch
Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides **two benefits**.

First, it allows you 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 you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

```
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.");
        }
        System.out.println("After catch statement.");
        }
}
```

This program generates the following **output:**
Division by zero.
After catch statement.

Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

Put differently, catch is not "called," so execution never "returns" to the try block from a catch. Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.
For example, in the next program each iteration of the for loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into a. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
        public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();
        for(int i=0; i<32000; i++) {
        try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
```

```
        } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
        }
        System.out.println("a: " + a);
        }
        }
}
```

throw

So far, you have only been catching exceptions that are **thrown by the Java run-time system.** However, it is possible for your program to throw an exception explicitly, using the throw statement. **You can throw exceptions yourself by using the throw statement.** The general form of throw is :

throw ThrowableInstance;

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 the 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.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

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

Here is the resulting output:
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

throws

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. You 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 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.

Example

```
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);
        }
}
}
```

Here is the output generated by running this example program:
inside throwOne
caught java.lang.IllegalAccessException: demo

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 you 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.

finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. **This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.**
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 {
        System.out.println("procC's 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.
If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

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
```

Multithreading

Unlike most other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains **two or more parts that can run concurrently.** Each part of such a program is called a **thread,** and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Multitasking threads require less overhead than multitasking processes. Processes are **heavyweight** tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.

Multithreading enables to write very efficient programs that make maximumn use of the CPU, because idle time can be kept to a minimum.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread class**, **its methods, and its companion interface, Runnable. To create a new thread, program will either extend Thread or implement the Runnable interface**. The Thread class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

Method	Meaning
getName	**Obtain a thread's name.**
getPriority	**Obtain a thread's priority.**
isAlive	**Determine if a thread is still running.**
join	**Wait for a thread to terminate.**
run	**Entry point for the thread.**
sleep	**Suspend a thread for a period of time.**
start	**Start a thread by calling its run method.**

fig. The methods in the thread class

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

■ It is the thread from which other "child" threads will be spawned.

■ Often it must be the last thread to finish execution because it performs various shutdown actions

Although the main thread is created automatically when program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. It general form is shown here:

static Thread currentThread( )

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

```
// Controlling the main Thread.
class CurrentThreadDemo {
        public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);
        try {
        for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
        }
```

```
        } catch (InterruptedException e) {
        System.out.println("Main thread interrupted");
        }
        }
}
```

Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

By default, the **name of the main thread is main. Its priority is 5**, which is the default value, and **main is also the name of the group of threads to which this thread belongs.**

The **sleep( ) method** causes the thread from which it is called **to suspend** execution **for the specified period of milliseconds**. Its general form is shown here:

static void sleep(long milliseconds) throws InterruptedException

The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException.

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called **run( ),** which is declared like this:

public void run( )

Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will **instantiate an object of type Thread** from within that class.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread. **In essence, start( ) executes a call to run( ).**

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {           // Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}

// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}

class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

The output produced by this program is as follows:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
```

```
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Passing this as the first argument indicates that you want the new thread to call the **run( )** method on this object. Next, **start( )** is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread's for loop to begin. **After calling start( ), NewThread's constructor returns to main( ).** When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three **child threads:**

```
// Create multiple threads.
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;

        NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        }

        // This is the entry point for thread.
        public void run() {
        try {
        for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
        System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
        }
        }
```

```
class MultiThreadDemo {
    public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
    try {
    // wait for other threads to end
    Thread.sleep(10000);
    } catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
    }
    }
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to sleep(10000) in main( ). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive( ) and join( )

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long

enough delay to ensure that all child threads terminate prior to the main thread.However, this is hardly a satisfactory solution. Fortunately, Thread provides a means by which you can answer this question. Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread.

The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

While isAlive( ) is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( ),** shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates.

Here is an improved version of the preceding example that uses join( ) to ensure that the main thread is the last to stop. It also demonstrates the isAlive( ) method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;
        NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        }
        // This is the entry point for thread.
        public void run() {
        try {
        for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
}
}

class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
```

```
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
```

Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

As you can see, after the calls to join( ) return, the threads have stopped executing.

Synchronization

When two or more threads need access to a **shared resource**, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. As you will see, Java provides unique, language-level support for it.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.** To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

// This program is not synchronized.

```
class Callme {
        void  call(String msg) {
        System.out.print("[" + msg);
        try {
        Thread.sleep(1000);
        } catch(InterruptedException e) {
        System.out.println("Interrupted");
        }
        System.out.println("]");
        }
}
class Caller implements Runnable {
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
        }
```

```
        public void run() {
        target.call(msg);
        }
}
class Synch {
        public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        // wait for threads to end
        try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
        } catch(InterruptedException e) {
        System.out.println("Interrupted");
        }
        }
}
```

Here is the output produced by this program:
Hello[Synchronized[World]
]

NOTE-Since the Call method is not synchronized more than one thread can access this method simultenously and result in the **Race Condition.**

To prevent from the race condition precede the call() with the keyword synchronized and analyze the output.

```
class Callme {
synchronized void call(String msg) {
.....
}
```

After synchronized has been added to call( ), the output of the program is as follows:
[Hello]
[Synchronized]
[World]

Suspending, Resuming, and Stopping Threads Using Java 2

wait()-method is invoked to suspend the execution of the thread.
notify()-method is invoked to wake up the thread.

```
// Suspending and resuming a thread for Java 2
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;
        boolean suspendFlag;
        NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
        try {
        for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
        while(suspendFlag) {
        wait();
        }
        }
        }
        } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
}
void mysuspend() {
        suspendFlag = true;
}
synchronized void myresume() {
        suspendFlag = false;
        notify();
        }
}
class SuspendResume {
        public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        try {
        Thread.sleep(1000);
        ob1.mysuspend();
```

```
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
            } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
            }
            // wait for threads to finish
            try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
            }
            System.out.println("Main thread exiting.");
            }
}
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
```

```
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

JAVA I/O STREAMS

Most programs use data in one form or another, whether it is as input, output, or both. The Java Development Kit (JDK) provides APIs for reading and writing streams of data. These APIs have been part of the core JDK since version 1.0, but are often overshadowed by the more well-known APIs, such as JavaBeans, RMI, JDBC, and so on. However, input and output streams are the backbone of the JDK APIs, and understanding them is not only crucial, but can also make programming with them a lot of fun.

To bring data into a program, a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. On the flip side, a program can open a stream to a data source and write to it in a serial fashion. Whether you are reading from a file or from a socket, the concept of serially reading from, and writing to different data sources is the same. For that very reason, once you understand the top level classes (java.io.Reader, java.io.Writer), the remaining classes are straightforward to work with.

Character Streams versus Byte Streams

Prior to JDK 1.1, the input and output classes (mostly found in the java.io package) only supported **8-bit byte streams**. The concept of **16-bit Unicode character** streams was introduced in JDK 1.1. While **byte streams** were supported via the **java.io.InputStream** and **java.io.OutputStream classes** and their subclasses, **character streams** are implemented by the **java.io.Reader and java.io.Writer classes** and their subclasses.

Most of the functionality available for byte streams is also provided for character streams. The

methods for character streams generally accept parameters of data type char parameters, while byte streams, you guessed it, work with byte data types. The names of the methods in both sets of classes are almost identical except for the suffix, that is, character-stream classes end with the suffix Reader or Writer and byte-stream classes end with the suffix InputStream and OutputStream. **For example, to read files using character streams, you would use the java.io.FileReader class; for reading it using byte streams you would use java.io.FileInputStream.**

Unless you are working with binary data, such as image and sound files, you should use readers and writers (character streams) to read and write information for the following reasons:

- They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
- They are easier to internationalize because they are not dependent upon a specific character encoding.
- They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

The Byte Stream Classes

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 12-1. *The Byte Stream Classes*

The Character Stream Classes

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 12-2. *The Character Stream I/O Classes*

Reading Console Input

In Java, console input is accomplished by reading from **System.in.** To obtain a character-based stream that is attached to the console, you wrap **System.in** in a **BufferedReader object**, to create a character stream.

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

Reading Characters

To read a character from a **BufferedReader**, use **read( ).** The version of read( ) that we will be using is

int read( ) throws IOException

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered. As you can see, it can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "q":

```
// Use a BufferedReader to read characters from the console.
import java.io.*;

class BRRead {
        public static void main(String args[])
        throws IOException
        {
        char c;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        // read characters
        do {
        c = (char) br.read();
        System.out.println(c);
        } while(c != 'q');
        }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

This output may look a little different from what you expected, because **System.in is line buffered, by default**. **This means that no input is actually passed to the program until you press ENTER.** As you can guess, this does not make read( ) particularly valuable for interactive, console input.

Reading Strings

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader class**. Its general form is shown here:

String readLine( ) throws IOException

As you can see, it returns a String object.

The following program demonstrates BufferedReader and the readLine( ) method to read from the console.It creates an array of String objects and then reads in lines of text, storing each line in the array. It will read up to 100 lines or until you enter "stop".

```
import java.io.*;
class TinyEdit {
```

```
        public static void main(String args[])
        throws IOException
        {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str[] = new String[100];
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        for(int i=0; i<100; i++) {
        str[i] = br.readLine();
        if(str[i].equals("stop")) break;
        }
        System.out.println("\nHere is your file:");
        // display the lines
        for(int i=0; i<100; i++) {
        if(str[i].equals("stop")) break;
        System.out.println(str[i]);
        }
        }
}
```

Here is a sample run:

```
Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
stop
Here is your file:
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
```

File I/O

The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. It contains two main abstract classess **InputStream and OutputStream**.

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an IOException on error conditions. Below are the list of the methods in InputStream :

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an IOException.
void mark(int numBytes)	Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ])	Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ], int offset, int numBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long numBytes)	Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.

OutputStream

OutputStream is an abstract class that defines streaming byte output. All of the methods in this class return a void value and throw an IOException in the case of errors. Table below shows the methods in OutputStream

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an IOException.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int b)	Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write( ) with expressions without having to cast them back to byte.
void write(byte buffer[ ])	Writes a complete array of bytes to an output stream.

void write(byte buffer[ ], int offset, int numBytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].

FileInputStream

The **FileInputStream class** creates an **InputStream** that you can use to read bytes from a file. Its two most common constructors are shown here:

FileInputStream(String filepath)
FileInputStream(File fileObj)

Either can throw a **FileNotFoundException.** Here, **filepath** is the full path name of a file, and **fileObj** is a File object that describes the file.

The following example creates two FileInputStreams that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("D:\abc.txt")
File f = new File("D:\abc.txt");
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allows us to closely examine the file using the File methods, before we attach it to an input stream. When a FileInputStream is created, it is also opened for reading.

The example below shows how to read a single byte, an array of bytes, and a subrange array of bytes. It also illustrates how to use **available( )** to determine the number of bytes remaining, and how to use the **skip( )** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory.

```
// Demonstrate FileInputStream.
import java.io.*;
class FileInputStreamDemo {
        public static void main(String args[]) throws Exception {
        int size;
        InputStream f = new FileInputStream("FileInputStreamDemo.java");
        System.out.println("Total Available Bytes: " + (size = f.available()));
        int n = size/40;
        System.out.println("First " + n + " bytes of the file one read() at a time");
        for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
        }
        System.out.println("\nStill Available: " + f.available());
        System.out.println("Reading the next " + n + " with one read(b[ ])");
        byte b[] = new byte[n];
        if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
        }
        System.out.println(new String(b, 0, n));
        System.out.println("\nStill Available: " + (size = f.available()));
```

```
        System.out.println("Skipping half of remaining bytes with skip()");
        f.skip(size/2);
        System.out.println("Still Available: " + f.available());
        System.out.println("Reading " + n/2 + " into the end of array");
        if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("couldn't read " + n/2 + " bytes.");
        }
        System.out.println(new String(b, 0, b.length));
        System.out.println("\nStill Available: " + f.available());
        f.close();
        }
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;
class FileInputS
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S
Still Available: 665
```

FileOutputStream

FileOutputStream creates an OutputStream that you can use to write bytes to a file. Its most commonly used constructors are shown here:

FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)

They can throw a FileNotFoundException or a SecurityException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, the file is opened in append mode. The fourth constructor was added by Java 2, version 1.4.

Creation of a FileOutputStream is not dependent on the file already existing. FileOutputStream will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an IOException will be thrown.

The following example creates a sample buffer of bytes by first making a String and then using the getBytes( ) method to extract the byte array equivalent. It then creates three files. The first, file1.txt, will contain every other byte from the sample. The second, file2.txt, will contain the entire set of bytes. The third and last, file3.txt, will contain only the last quarter. Unlike the FileInputStream methods, all of the FileOutputStream methods have a return type of void. In the case of an error, these methods will throw an IOException

// Demonstrate FileOutputStream.

```
import java.io.*;
class FileOutputStreamDemo {
        public static void main(String args[]) throws Exception {
        String source = "Now is the time for all good men\n"
        + " to come to the aid of their country\n"
        + " and pay their due taxes.";
        byte buf[] = source.getBytes();
        OutputStream f0 = new FileOutputStream("file1.txt");
        for (int i=0; i < buf.length; i += 2) {
        f0.write(buf[i]);
        }
        f0.close();
        OutputStream f1 = new FileOutputStream("file2.txt");
        f1.write(buf);
        f1.close();
        OutputStream f2 = new FileOutputStream("file3.txt");
        f2.write(buf,buf.length-buf.length/4,buf.length/4);
        f2.close();
        }
}
```

Here are the contents of each file after running this program.

First, file1.txt:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, file2.txt:

```
Now is the time for all good men
to come to the aid of their country
and pay their due taxes.
```

Finally, file3.txt:

```
nd pay their due taxes.
```

The Character Streams

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with **Unicode characters**. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters.At the top of the **character stream hierarchies** are the **Reader**

and **Writer** abstract classes.

Reader

Reader is an abstract class that defines Java's model of streaming character input. All of the methods in this class will throw an IOException on error conditions. Table below provides a synopsis of the methods in Reader.

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.

Table 17-3. *The Methods Defined by* Reader

FileReader

The FileReader class creates a Reader that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

FileReader(String filePath)

FileReader(File fileObj)
Either can throw a **FileNotFoundException**. Here, **filePath** is the full path name of a file, and **fileObj** is a File object that describes the file.

The following example shows how to read lines from a file and print these to the standard output stream. It reads its own source file, which must be in the current directory.

// Demonstrate FileReader.

```
import java.io.*;
class FileReaderDemo {
public static void main(String args[]) throws Exception {
FileReader fr = new FileReader("FileReaderDemo.java");
BufferedReader br = new BufferedReader(fr);
String s;
while((s = br.readLine()) != null) {
System.out.println(s);
}
fr.close();
}
}
```

Writer
Writer is an abstract class that defines streaming character output. All of the methods in this class return a void value and throw an IOException in the case of errors. Table below shows a synopsis of the methods in Writer.

Method	Description
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with expressions without having to cast them back to **char**.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *str*, beginning at the specified *offset*.

Table 17-4. *The Methods Defined by* Writer

FileWriter

FileWriter creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

FileWriter(String filePath)
FileWriter(String filePath, boolean append)
FileWriter(File fileObj)
FileWriter(File fileObj, boolean append)

They can throw an IOException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, then output is appended to the end of the file.

Creation of a FileWriter is not dependent on the file already existing. FileWriter will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an IOException will be thrown.The following example is a character stream version of an example shown earlier when FileOutputStream was discussed.

This version creates a sample buffer of characters by first making a String and then using the **getChars( )** method to extract the character array equivalent. It then creates three files. The first, **file1.txt**, will contain every other character from the sample. The second, **file2.txt**, will contain the entire set of characters. Finally, the third, **file3.txt**, will contain only the last quarter.

// **Demonstrate FileWriter.**

```
import java.io.*;
class FileWriterDemo {
public static void main(String args[]) throws Exception {
        String source = "Now is the time for all good men\n"
        + " to come to the aid of their country\n"
        + " and pay their due taxes.";
        char buffer[] = new char[source.length()];
        source.getChars(0, source.length(), buffer, 0);
        FileWriter f0 = new FileWriter("file1.txt");
        for (int i=0; i < buffer.length; i += 2) {
        f0.write(buffer[i]);
        }
        f0.close();
        FileWriter f1 = new FileWriter("file2.txt");
        f1.write(buffer);
        f1.close();
        FileWriter f2 = new FileWriter("file3.txt");
        f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
        f2.close();
        }
}
```

User Interface components with swing

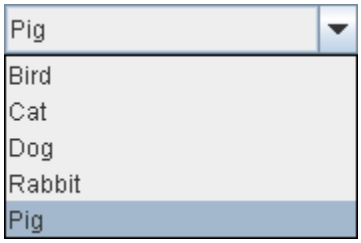
A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application. A GUI **(pronounced "GOO-ee")** gives an application a distinctive "look and feel." GUIs are built from GUI components. These are sometimes called **controls or widgets**—short for window gadgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. The Swing GUI components are defined in the **javax.swing package**.

Different Java GUI controls

Basic Controls

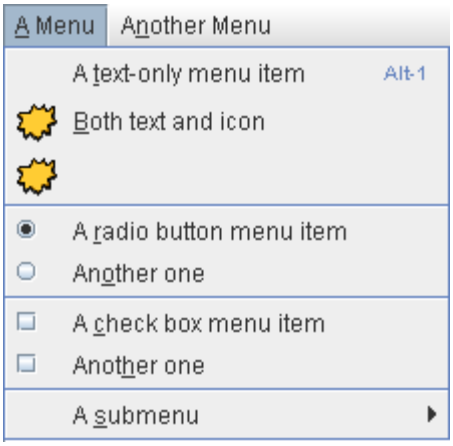
JButton

JCheckBox



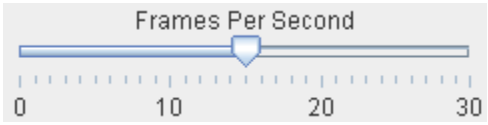
JComboBox

JList



JMenu

JRadioButton

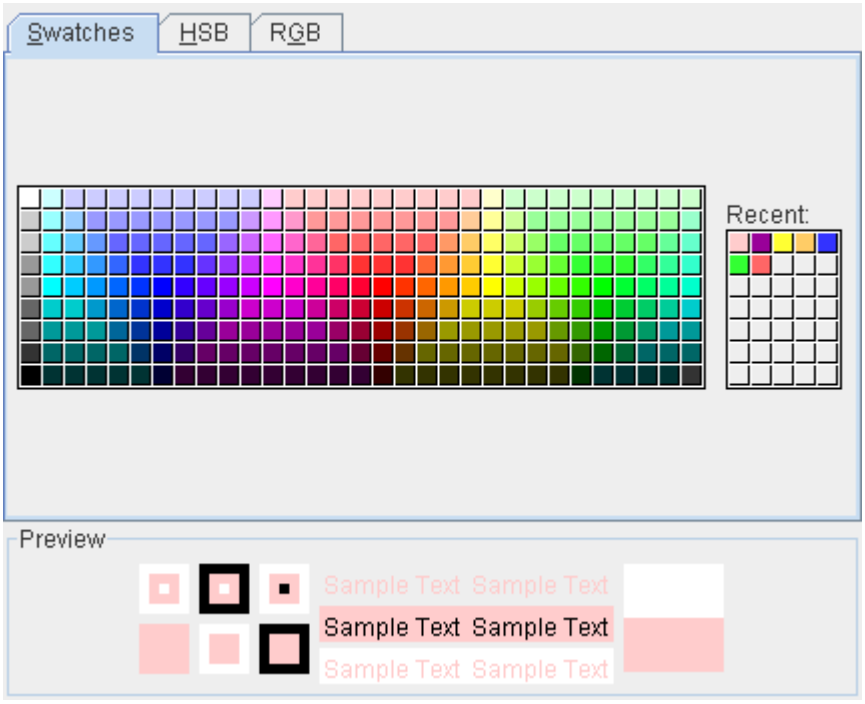


JSlider

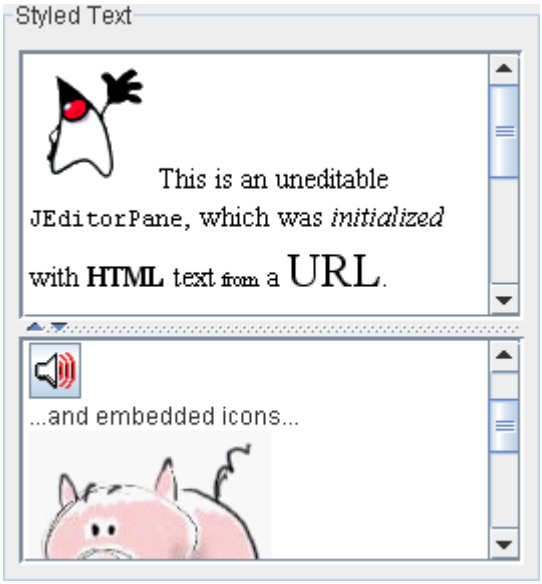
JTextField

JPasswordField

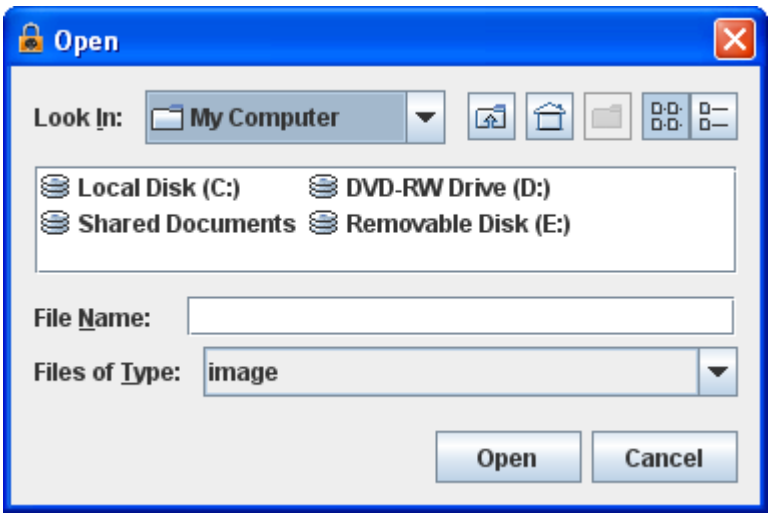
Interactive Controls



JColorChooser



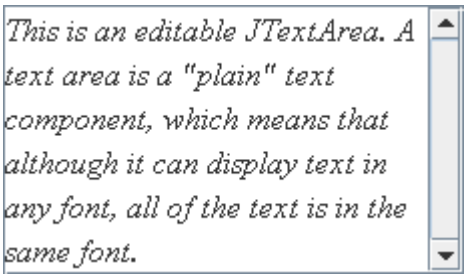
JEditorPane



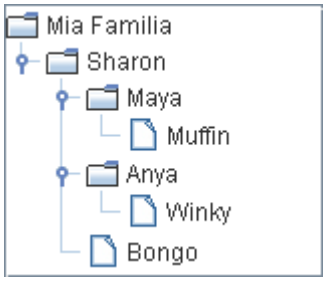
JFileChooser

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

JTable



JTextArea



JTree

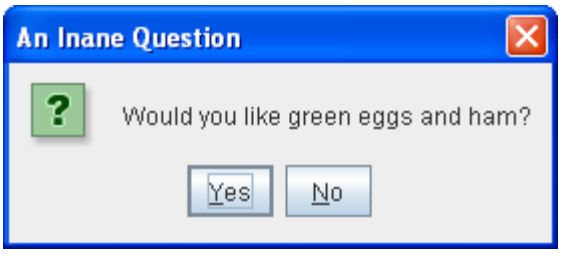
<u>Uneditable Controls</u>

Jlabel

JProgressBar

JToolTip

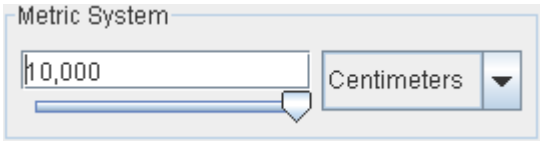
Top-Level Containers



JDialog

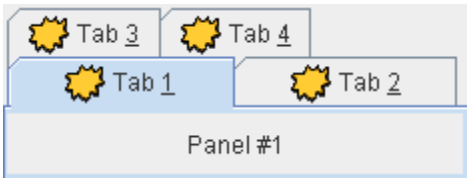
JFrame

General Purpose Containers

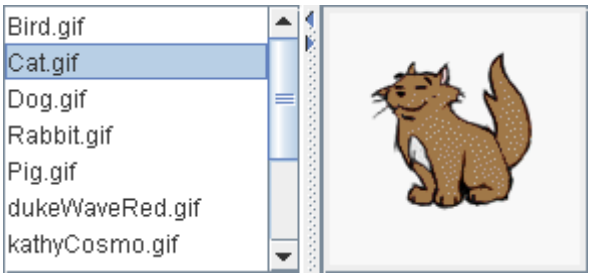


JPanel

JScrollPane



JTabbedPane



JSplitPane

Pls Visit the URL http://docs.oracle.com/javase/tutorial/uiswing/components/index.html for detailed study of Java GUI using Swing.

Overview of Swing Components

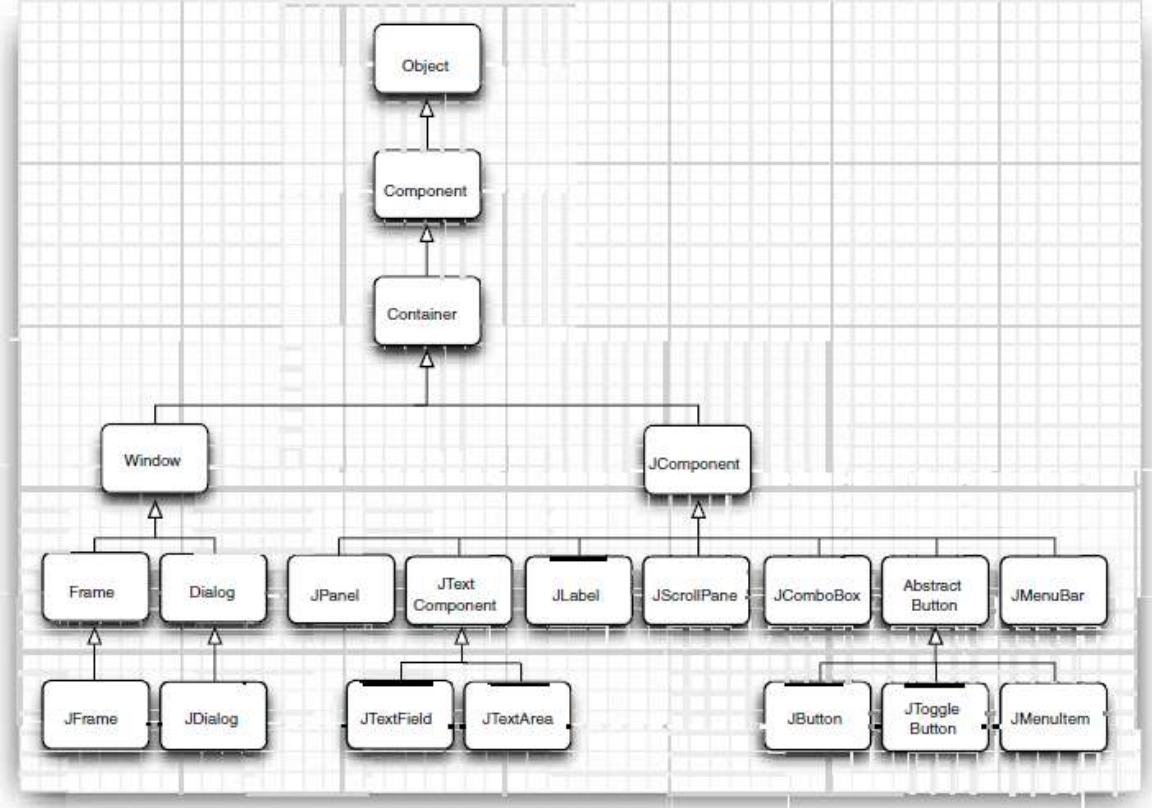


Fig 3. Inheritance hierarchy for the Component class

Component	Description
`JLabel`	Displays uneditable text and/or icons.
`JTextField`	Typically receives input from the user.
`JButton`	Triggers an event when clicked with the mouse.
`JCheckBox`	Specifies an option that can be selected or not selected.
`JComboBox`	A drop-down list of items from which the user can make a selection.
`JList`	A list of items from which the user can make a selection by clicking on any one of them. Multiple elements can be selected.
`JPanel`	An area in which components can be placed and organized.

fig. Some basic GUI components

Swing vs. AWT

There are actually two sets of Java GUI components. In Java's early days, GUIs were built with components from the **Abstract Window Toolkit (AWT)** in **package java.awt.** These look like the native GUI components of the platform on which a Java program executes. For example, a Button object displayed in a Java program running on Microsoft Windows looks like those in other Windows

applications. On Apple Mac OS X, the Button looks like those in other Mac applications. Sometimes, even the manner in which a user can interact with an AWT component differs between platforms. The component's appearance and the way in which the user interacts with it are known as its **look-and-feel**.

Swing GUI components allow you to specify a uniform look-and-feel for your application across all platforms or to use each platform's custom look-and-feel. An application can even change the look-and-feel during execution to enable users to choose their own preferred look-and-feel.

Most Swing components are **lightweight components**—they're written, manipulated and displayed completely in Java. AWT components are **heavyweight components**, because they rely on the local platform's windowing system to determine their functionality and their look-and-feel. Several Swing components are heavyweight components.

Java Top-Level Containers

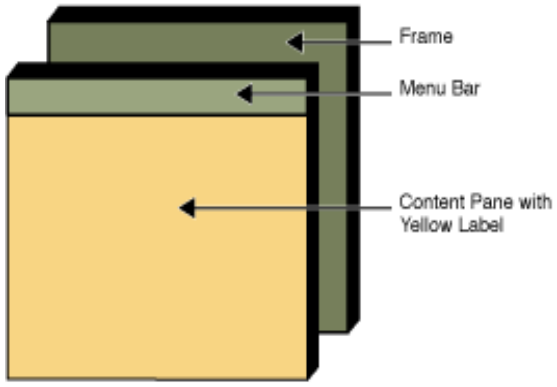
Swing provides three generally useful **top-level container classes**: **JFrame, JDialog, and JApplet.** When using these classes, you should keep these facts in mind:

1. To appear onscreen, every GUI component must be part of a **containment hierarchy**. A containment hierarchy is a tree of components that has a **top-level container as its root**.

2. Each GUI component can be contained only once. If a component is already in a container and you try to add it to another container, the component will be removed from the first container and then added to the second.

3. Each top-level container has a **content pane** that, generally speaking, contains (directly or indirectly) the visible components in that top-level container's GUI.

4. You can optionally add a **menu bar** to a top-level container. The menu bar is by convention positioned within the top-level container, but outside the content pane.



Top-Level Containers and Containment Hierarchies

Each program that uses Swing components has **at least one top-level container**. This top-level container is the **root** of a **containment hierarchy** — the hierarchy that contains all of the Swing components that appear inside the top-level container.

As a rule, **a standalone application with a Swing-based GUI** has at least one containment hierarchy with **a JFrame as its root**. For example, if an application has **one main window** and **two dialogs**, then the application has **three containment hierarchies**, and thus **three top-level containers. One** containment hierarchy has a **JFrame as its root,** and **each of the other two** has a **JDialog object as its root**.

A Swing-based applet has at least one containment hierarchy, exactly one of which is rooted by a **JApplet** object. For example, an applet that brings up a dialog has two containment hierarchies. The components in the browser window are in a containment hierarchy rooted by a JApplet object. The dialog has a containment hierarchy rooted by a JDialog object.

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

public class TopLevelDemo {
    private static void createAndShowGUI() {
     //Create and set up the window.
     JFrame frame = new JFrame("TopLevelDemo");
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     //Create the menu bar.  Make it have a green background.
     JMenuBar greenMenuBar = new JMenuBar();
     greenMenuBar.setOpaque(true);
     greenMenuBar.setBackground(new Color(154, 165, 127));
     greenMenuBar.setPreferredSize(new Dimension(200, 20));

     //Create a yellow label to put in the content pane.
     JLabel yellowLabel = new JLabel();
     yellowLabel.setOpaque(true);
     yellowLabel.setBackground(new Color(248, 213, 131));
     yellowLabel.setPreferredSize(new Dimension(200, 180));

     //Set the menu bar and add the label to the content pane.
     frame.setJMenuBar(greenMenuBar);
     frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);

 /*
//Create a panel and add components to it.
JPanel contentPane = new JPanel(new BorderLayout());
contentPane.setBorder(someBorder);
contentPane.add(someComponent, BorderLayout.CENTER);
contentPane.add(anotherComponent, BorderLayout.PAGE_END);
topLevelContainer.setContentPane(contentPane);
   */
```

```
 //Display the window.
    frame.pack();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    //Schedule a job for the event-dispatching thread:
    //creating and showing this application's GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
  }
}         //end of class TopLevelDemo
```

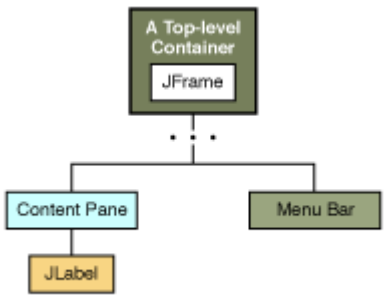


fig. the containment hierarchy for this example's GUI

Creating a Frame

A top-level window (that is, a window that is not contained inside another window) is called a **frame** in Java. The AWT library has a class, called Frame, for this top level. The Swing version of this class is called JFrame and extends the Frame class.

```
import javax.swing.*;

 public class SimpleFrameTest
 {
 public static void main(String[] args)
 {
 SimpleFrame frame = new SimpleFrame();
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.setVisible(true);
}
 }

class SimpleFrame extends JFrame
```

```
{
 public SimpleFrame()
 {
setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
 }
 public static final int DEFAULT_WIDTH = 300;
 public static final int DEFAULT_HEIGHT = 200;
 }
```

fig. A simple frame

Swing GUI components are instances of class JFrame or a subclass of JFrame. JFrame is an indirect subclass of class java.awt.Window that provides the basic attributes and behaviors of a window—a title bar at the top, and buttons to minimize, maximize and close the window.

Buttons, text fields, and other user interface elements extend the class Component. Components can be placed inside containers such as panels.

By default, closing a window simply hides the window. However, when the user closes the frame, we would like the application to terminate. **setDefaultCloseOperation method(inherited from class JFrame) with constant JFrame.EXIT_ON_CLOSE** as the argument indicate that the program should terminate when the window is closed by the user.

Dialog Box Using Swing

Dialog boxes are windows in which programs display important messages to the user or obtain informationfrom the user. Most applications you use on a daily basis use windows or dialog boxes (also called **dialogs**) to interact with the user.

Java's **JOptionPane class (package javax.swing)** provides prebuilt dialog boxes for both input and output. These are displayed by invoking **static JOptionPane methods**. Program below presents a simple addition application that uses two input dialogs to obtain integers from the user and a message dialog to display the sum of the integers the user enters.

// Fig. 14.2: Addition.java

// 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
```

Output

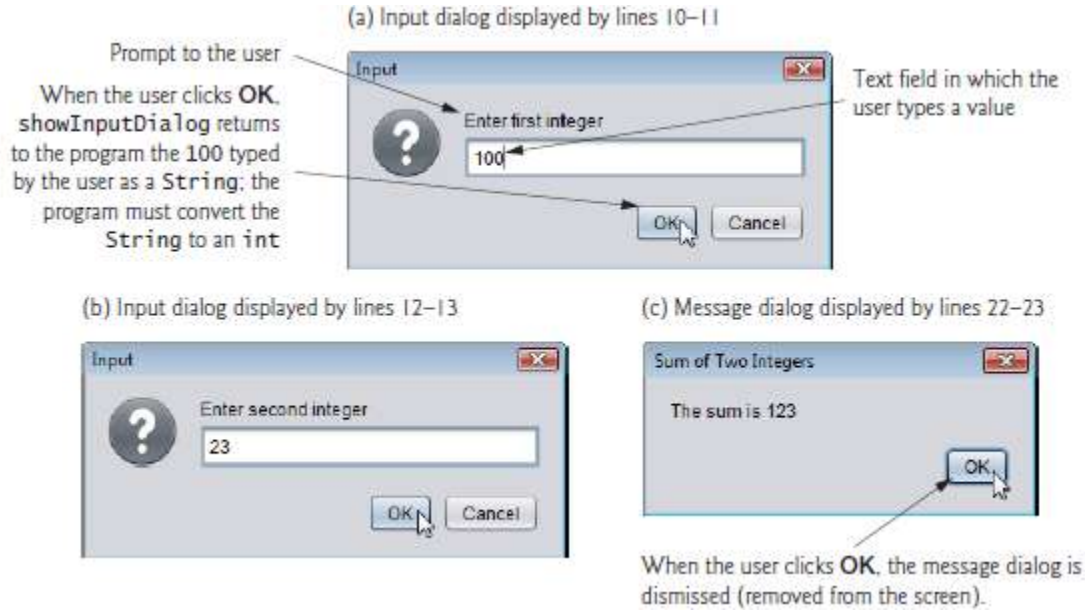


fig 1

Input Dialogs

Line 3 imports class JOptionPane. Lines 10–11 declare the local String variable first- Number and assign it the result of the call to **JOptionPane static method showInputDialog**. This method displays an input dialog using the method's String argument ("Enter first integer") as a prompt.

Message Dailogs

Lines 22–23 use JOptionPane static method showMessageDialog to display a message dialog (the last screen of Fig. 1) containing the sum. The **first argument** helps the Java application determine **where to**

position the dialog box. A dialog is typically displayed from a GUI application with its own window. The first argument refers to that window (known as the parent window) and causes the dialog to appear centered over the parent. If the first argument is null, the dialog box is displayed at the center of your screen. The **second argument** is the message to display—in this case, **the result of concatenating the String "The sum is " and the value of sum**. The **third argument—"Sum of Two Integers"—is the String that should appear in the title bar at the top of the dialog. The fourth argument—JOptionPane.PLAIN_MESSAGE—is the type ofmessage dialog to display.** A PLAIN_MESSAGE dialog does not display an icon to the left of the message.

JOptionPane Message Dialog Constants

The constants that represent the message dialog types are shown in Fig. 2. All message dialog types except PLAIN_MESSAGE display an icon to the left of the message. These icons provide a visual indication of the message's importance to the user. A QUESTION_MESSAGE icon is the default icon for an input dialog box (see Fig. 2).

Message dialog type	Icon	Description
ERROR_MESSAGE		Indicates an error.
INFORMATION_MESSAGE		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.

fig. 2

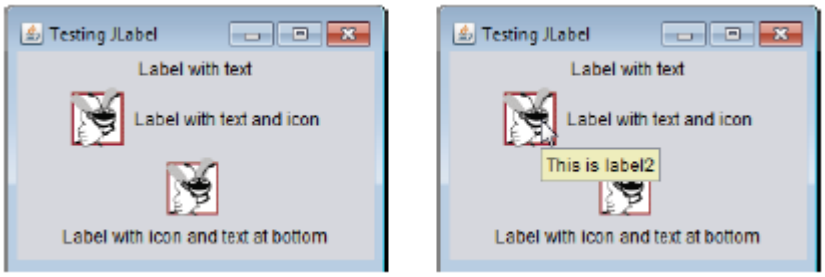
JLabel

A typical GUI consists of many components. GUI designers often provide text stating the purpose of each components. Such text is known as a **label** and is created with a **JLabel**—a subclass of JComponent. **A JLabel displays read-only text, an image, or both text and an image.** Applications rarely change a label's contents after creating it.

```
// LabelFrame.java
// Demonstrating the JLabel class.
 import java.awt.FlowLayout;              // specifies how components are arranged
 import javax.swing.JFrame;               // provides basic window features
 import javax.swing.JLabel;               // displays text and images
 import javax.swing.SwingConstants;       // common constants used with Swing
 import javax.swing.Icon;                 // interface used to manipulate images
 import javax.swing.ImageIcon;            // loads images
```

```
public class LabelFrame extends JFrame
{
private JLabel label1;                    // JLabel with just text
private JLabel label2;                    // JLabel constructed with text and icon
private JLabel label3;                    // JLabel with added text and icon

// LabelFrame constructor adds JLabels to JFrame
public LabelFrame()
{
super( "Testing JLabel" );
setLayout( new FlowLayout() );            // set frame layout

// JLabel constructor with a string argument
label1 = new JLabel( "Label with text" );
label1.setToolTipText( "This is label1" );
add( label1 ); // add label1 to JFrame

// JLabel constructor with string, Icon and alignment arguments
Icon bug = new ImageIcon( getClass().getResource( "bug1.png" ) );
label2 = new JLabel( "Label with text and icon", bug,
SwingConstants.LEFT );
label2.setToolTipText( "This is label2" );
add( label2 );                            // add label2 to JFrame

label3 = new JLabel();                    // JLabel constructor no arguments
label3.setText( "Label with icon and text at bottom" );
label3.setIcon( bug );                    // add icon to JLabel
label3.setHorizontalTextPosition( SwingConstants.CENTER );
label3.setVerticalTextPosition( SwingConstants.BOTTOM );
label3.setToolTipText( "This is label3" );
add( label3 );                            // add label3 to JFrame
} // end LabelFrame constructor
} // end class LabelFrame

// LabelTest.java
// Testing LabelFrame.
import javax.swing.JFrame;

public class LabelTest
{
public static void main( String[] args )
{
LabelFrame labelFrame = new LabelFrame();          // create LabelFrame
labelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
labelFrame.setSize( 260, 180 );                    // set frame size
labelFrame.setVisible( true );                     // display frame
} // end main
} // end class LabelTest
```

output



Introduction to event handling

Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the Send button tells the application to send the e-mail to the specified e-mail addresses. **GUIs are event driven**. When the user interacts with a GUI component, the interaction—known as an **event—drives the program to perform a task**. Some common user interactions that cause an application to perform a task include **clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse.** The code that performs a task in response to an event is called an **event handler**, and the overall process of responding to events is known as **event handling**.

Steps Required to Set Up Event Handling for a GUI Component

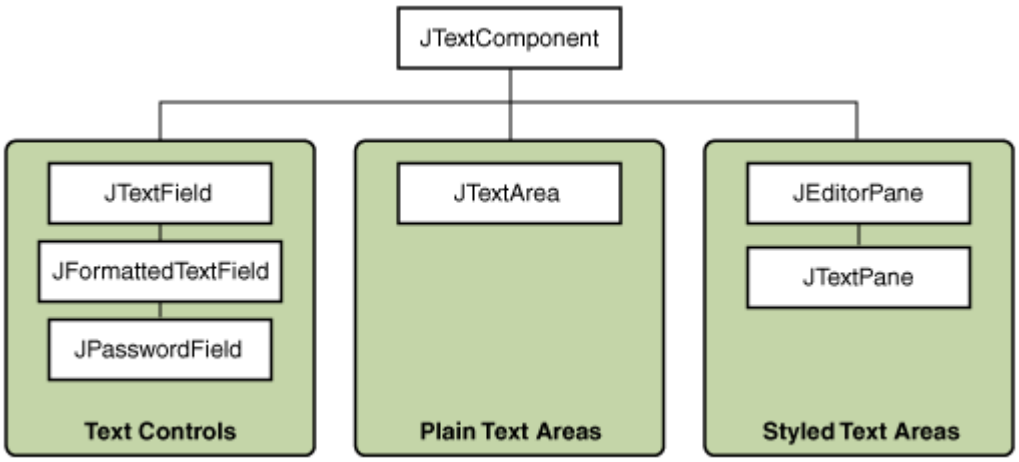
Before an application can respond to an event for a particular GUI component, you must:

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface.**

2. Indicate that an object of the class from Step 1 should be notified when the event occurs—known as **registering the event handler.**

Text Field with event handling

A **text field** is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an **action event**. If you need to obtain **more than one line** of input from the user, use a **text area.**

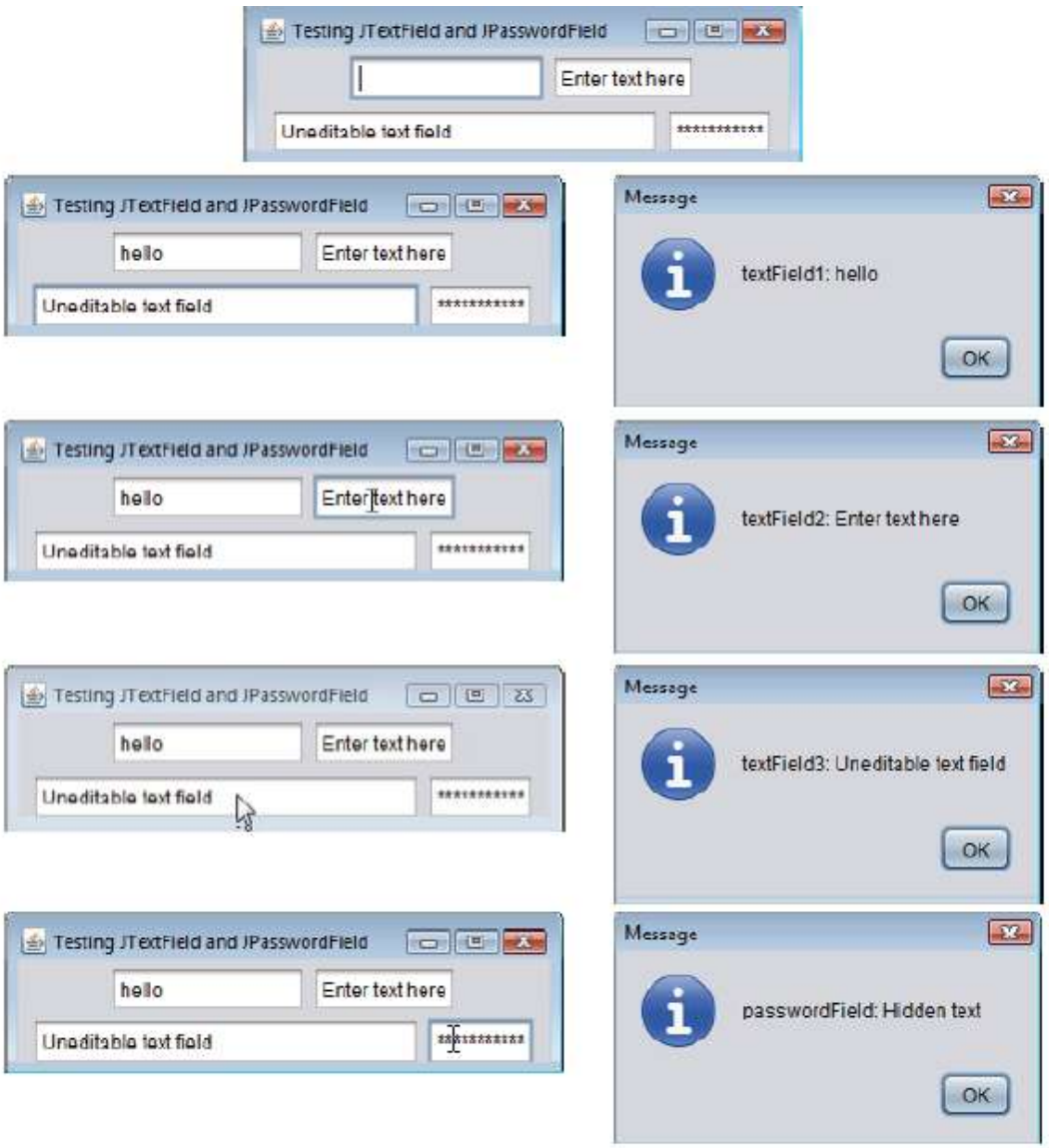



```
// Demonstrating the JTextField class.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JPasswordField;
import javax.swing.JOptionPane;

 public class TextFieldFrame extends JFrame
 {
 private JTextField textField1; // text field with set size
private JTextField textField2; // text field constructed with text
 private JTextField textField3; // text field with text and size
 private JPasswordField passwordField; // password field with text

 // TextFieldFrame constructor adds JTextFields to JFrame
 public TextFieldFrame()
 {
super( "Testing JTextField and JPasswordField" );
setLayout( new FlowLayout() ); // set frame layout

// construct textfield with 10 columns
textField1 = new JTextField( 10 );
add( textField1 ); // add textField1 to JFrame

// construct textfield with default text
textField2 = new JTextField( "Enter text here" );
add( textField2 ); // add textField2 to JFrame

// construct textfield with default text and 21 columns
textField3 = new JTextField( "Uneditable text field", 21 );
textField3.setEditable( false ); // disable editing
add( textField3 ); // add textField3 to JFrame
```

```
// construct passwordfield with default text
passwordField = new JPasswordField( "Hidden text" );
add( passwordField ); // add passwordField to JFrame

 // register event handlers
TextFieldHandler handler = new TextFieldHandler();
textField1.addActionListener( handler );
textField2.addActionListener( handler );
textField3.addActionListener( handler );
passwordField.addActionListener( handler );

 } // end TextFieldFrame constructor

 // private inner class for event handling
private class TextFieldHandler implements ActionListener
{
 // process text field events
public void actionPerformed( ActionEvent event )
 {
 String string = ""; // declare string to display

// user pressed Enter in JTextField textField1
 if(event.getSource() == textField1)
string = String.format( "textField1: %s",event.getActionCommand());

 // user pressed Enter in JTextField textField2
else if(event.getSource() == textField2 )
string = String.format( "textField2: %s",event.getActionCommand() );

// user pressed Enter in JTextField textField3
else if(event.getSource() == textField3 )
 string = String.format( "textField3: %s",event.getActionCommand());

 // user pressed Enter in JTextField passwordField
 else if(event.getSource() == passwordField )
string = String.format( "passwordField: %s",event.getActionCommand() );

 // display JTextField content
JOptionPane.showMessageDialog( null, string );
 } // end method actionPerformed
 } // end private inner class TextFieldHandler
 } // end class TextFieldFrame


 // Testing TextFieldFrame.
import javax.swing.JFrame;
```

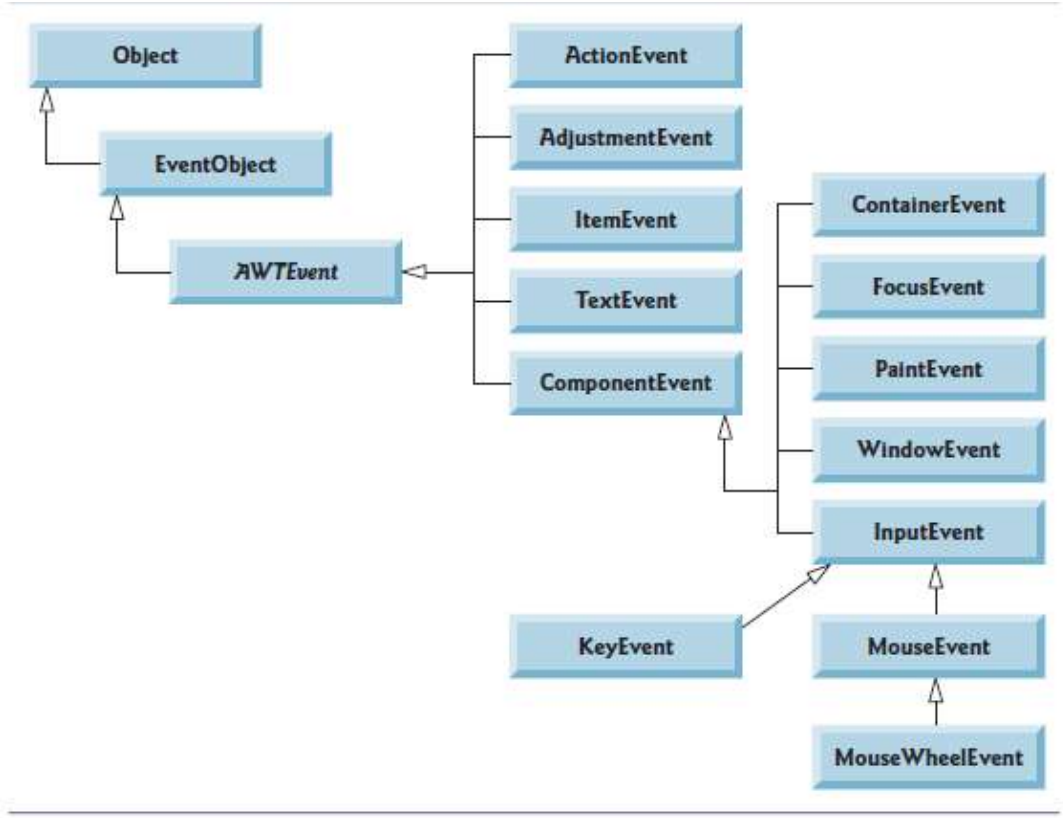
```
public class TextFieldTest
 {
public static void main( String[] args )
 {
TextFieldFrame textFieldFrame = new TextFieldFrame();
 textFieldFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 textFieldFrame.setSize( 350, 100 ); // set frame size
textFieldFrame.setVisible( true ); // display frame
 } // end main
} // end class TextFieldTest
```

output



Common GUI Event Types and Listener Interfaces

Many different types of events can occur when the user interacts with a GUI. The event information is stored in an object of a class that extends **AWTEvent (from package java.awt).** Figure below illustrates a hierarchy containing many event classes from the package **java.awt.event**. These event types are used with both AWT and Swing components. Additional event types that are specific to Swing GUI components are declared in package **javax.swing.event.**



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 super class 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. (Added by Java 2, version 1.4)
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.

fig. Some event classes of package java.awt.event

The event-handling mechanism mainly consist of three parts—**the event source, the event object and the event listener.** The **event source** is the GUI component with which the user interacts. The **event object** encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The **event listener** is an object that's notified by the event source when an event occurs; in effect, it "listens" for an event, and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. This event-handling model is known as the **delegation event model**—an event's processing is delegated to an object (the event listener) in the application.

Event Source	Description
Button	Generates action events when the button is pressed.
Checkbox	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.
Scrollbar	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 [redacted] *Event Source Examples*

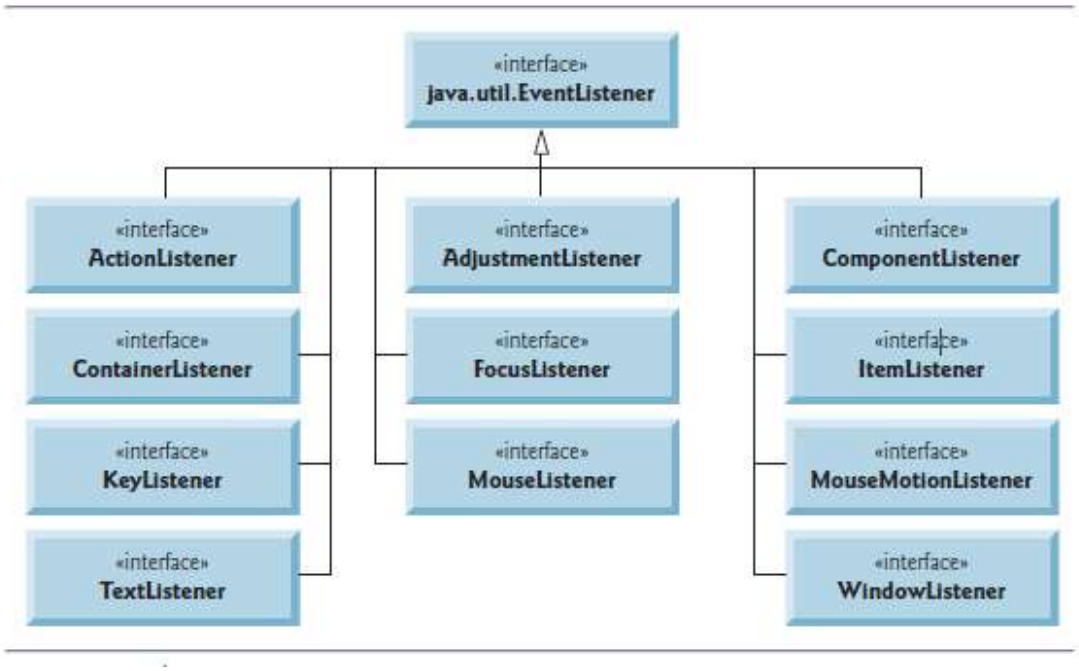
fig. Event Source Examples

For each event-object type, there's typically a **corresponding event-listener interface**. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from

packages java.awt.event and javax.swing.event. Many of the event-listener types are common to both Swing and AWT components. Such types are declared in package java.awt.event, and some of them are shown in Fig. below. Additional event-listener types that are specific to Swing components are declared in package javax.swing.event.

Each event-listener interface specifies one or more event-handling methods that must be declared in the class that implements the interface.**Any class which implements an interface must declare all the abstract methods of that interface;** otherwise, the class is an abstract class and cannot be used to create objects.

When an event occurs, the GUI component with which the user interacted **notifies its registered listeners by calling each listener's appropriate event-handling method**. For example, when the user presses the Enter key in a JTextField, the registered listener's actionPerformed method is called.



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. (Added by Java 2, version 1.4)
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. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table *Commonly Used Event Listener Interfaces*

fig. Some common event-listener interfaces of package java.awt.event

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

Note *The AWT processes the resize and move events. The **componentResized( )** and **componentMoved( )** methods are provided for notification purposes only.*

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

JCheckBox

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here.

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

MouseWheelListener was added by Java 2, version 1.4.

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

```
void textChanged(TextEvent te)
```

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or losses input focus. Their general forms are shown here.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

WindowFocusListener was added by Java 2, version 1.4.

The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

JButton

A button is a component **the user clicks to trigger a specific action**. A Java application can use several types of buttons, including **command buttons, checkboxes, toggle buttons and radio buttons.** Figure below shows the inheritance hierarchy of the **Swing buttons** . As you can see, all the button types are subclasses of AbstractButton (package javax.swing), which declares the common features of Swing buttons.

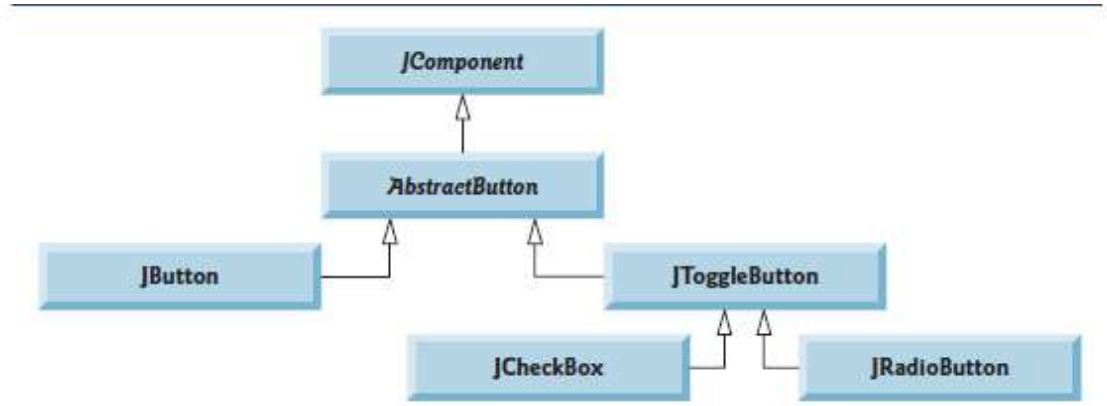


fig. Swing button hierarchy.

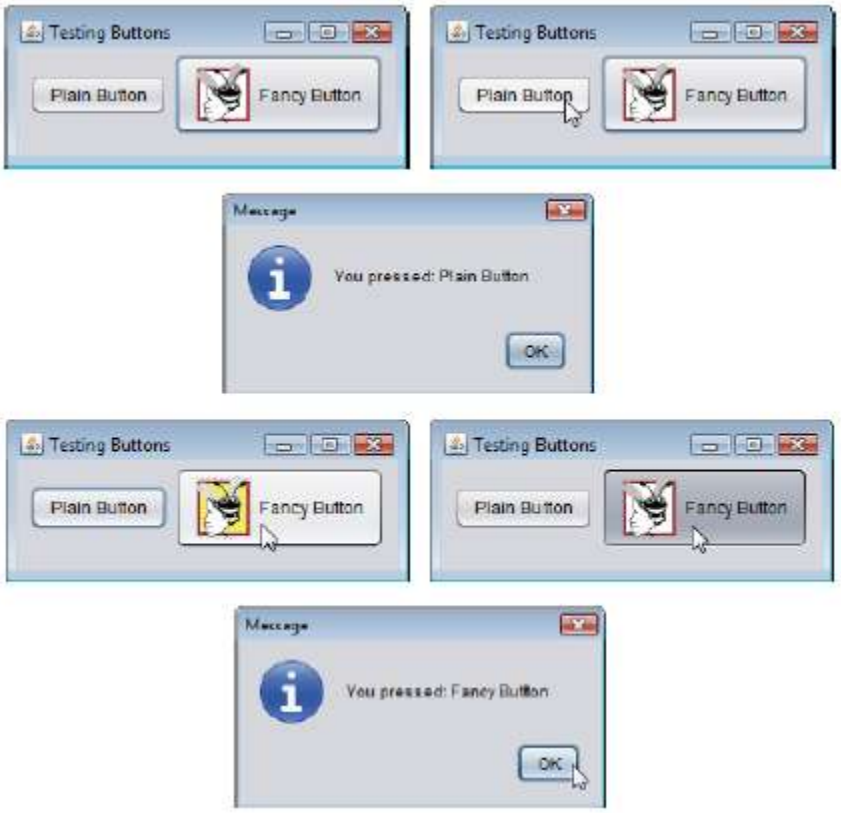
A command button generates an ActionEvent when the user clicks it. Command buttons are created with class **JButton**. The text on the face of a JButton is called a **button label**. A GUI can have many JButtons, but each button label should be unique in the portion of the GUI that's currently displayed.

The application below creates two JButtons and demonstrates that JButtons support the display of Icons. Event handling for the buttons is performed by a single instance of inner class ButtonHandler.

```
// ButtonFrame.java
// Creating JButtons.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class ButtonFrame extends JFrame
{
private JButton plainJButton; // button with just text
private JButton fancyJButton; // button with icons

// ButtonFrame adds JButtons to JFrame
public ButtonFrame()
{
super( "Testing Buttons" );
setLayout( new FlowLayout() ); // set frame layout

plainJButton = new JButton( "Plain Button" ); // button with text
add( plainJButton ); // add plainJButton to JFrame
```

```
Icon bug1 = new ImageIcon( getClass().getResource( "bug1.gif" ) );
Icon bug2 = new ImageIcon( getClass().getResource( "bug2.gif" ) );
fancyJButton = new JButton( "Fancy Button", bug1 ); // set image
fancyJButton.setRolloverIcon( bug2 ); // set rollover image
add( fancyJButton ); // add fancyJButton to JFrame

// create new ButtonHandler for button event handling
ButtonHandler handler = new ButtonHandler();
fancyJButton.addActionListener( handler );
plainJButton.addActionListener( handler );
 } // end ButtonFrame constructor

// inner class for button event handling
private class ButtonHandler implements ActionListener
 {
 // handle button event
public void actionPerformed( ActionEvent event )
 {
JOptionPane.showMessageDialog(ButtonFrame.this , String.format(
 "You pressed: %s",event.getActionCommand() ));
 } // end method actionPerformed
 } // end private inner class ButtonHandler
 } // end class ButtonFrame


// ButtonTest.java
 // Testing ButtonFrame.
import javax.swing.JFrame;

public class ButtonTest
 {
 public static void main( String[] args )
 {
 ButtonFrame buttonFrame = new ButtonFrame(); // create ButtonFrame
 buttonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 buttonFrame.setSize( 275, 110 ); // set frame size
 buttonFrame.setVisible( true ); // display frame
 } // end main
 } // end class ButtonTest
```

output



JCheckBox

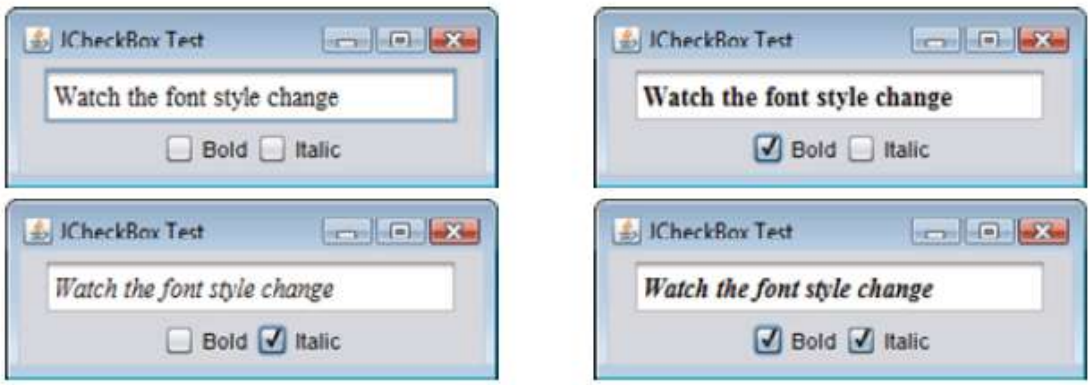
Classes **JCheckBox** and **JRadioButton** are subclasses of JToggleButton. A JRadioButton is different from a JCheckBox in that normally several JRadioButtons are grouped together and are mutually exclusive—only one in the group can be selected at any time where as more than one check box can be selected at a time.

```
//CheckBoxFrame.java
// Creating JCheckBox buttons.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JCheckBox;

public class CheckBoxFrame extends JFrame
 {
        private JTextField textField; // displays text in changing fonts
        private JCheckBox boldJCheckBox; // to select/deselect bold
        private JCheckBox italicJCheckBox; // to select/deselect italic

 // CheckBoxFrame constructor adds JCheckBoxes to JFrame
        public CheckBoxFrame()
```

```
{
        super( "JCheckBox Test" );
        setLayout( new FlowLayout() ); // set frame layout

        // set up JTextField and set its font
        textField = new JTextField( "Watch the font style change", 20 );
        textField.setFont( new Font( "Serif", Font.PLAIN, 14 ) );
        add( textField ); // add textField to JFrame

        boldJCheckBox = new JCheckBox( "Bold" ); // create bold checkbox
        italicJCheckBox = new JCheckBox( "Italic" ); // create italic
        add( boldJCheckBox ); // add bold checkbox to JFrame
        add( italicJCheckBox ); // add italic checkbox to JFrame

        // register listeners for JCheckBoxes
        CheckBoxHandler handler = new CheckBoxHandler();
        boldJCheckBox.addItemListener( handler );
        italicJCheckBox.addItemListener( handler );

        } // end CheckBoxFrame constructor

        // private inner class for ItemListener event handling
        private class CheckBoxHandler implements ItemListener
        {
        // respond to checkbox events
        public void itemStateChanged( ItemEvent event )
        {
                Font font = null; // stores the new Font
                // determine which CheckBoxes are checked and create Font
                if (boldJCheckBox.isSelected() && italicJCheckBox.isSelected() )
                font = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
                else if( boldJCheckBox.isSelected())
                font = new Font( "Serif", Font.BOLD, 14 );
                else if( italicJCheckBox.isSelected())
                font = new Font( "Serif", Font.ITALIC, 14 );
                else
                font = new Font( "Serif", Font.PLAIN, 14 );
                textField.setFont( font ); // set textField's font
        } // end method itemStateChanged
    } // end private inner class CheckBoxHandler
} // end class CheckBoxFrame

// CheckBoxTest.java
// Testing CheckBoxFrame.
import javax.swing.JFrame;

public class CheckBoxTest
{
```

```
        public static void main( String[] args )
        {
                CheckBoxFrame checkBoxFrame = new CheckBoxFrame();
                checkBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
                checkBoxFrame.setSize( 275, 100 ); // set frame size
                checkBoxFrame.setVisible( true ); // display frame
        } // end main
} // end class CheckBoxTest
```

Output



JRadioButton

Radio buttons (declared with class JRadioButton) are similar to checkboxes in that they have **two states—selected and not selected (also called deselected**). However, **radio buttons** normally appear as a group **in which only one button can be selected at a time** .Selecting a different radio button forces all others to be deselected. **Radio buttons are used to represent mutually exclusive options (i.e., multiple options in the group cannot be selected at the same time).**

```
// Fig. 14.19: RadioButtonFrame.java
// Creating radio buttons using ButtonGroup and JRadioButton.
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;

 public class RadioButtonFrame extends JFrame
 {
private JTextField textField; // used to display font changes
private Font plainFont; // font for plain text
```

```
private Font boldFont; // font for bold text
private Font italicFont; // font for italic text
private Font boldItalicFont; // font for bold and italic text
private JRadioButton plainJRadioButton; // selects plain text
private JRadioButton boldJRadioButton; // selects bold text
private JRadioButton italicJRadioButton; // selects italic text
private JRadioButton boldItalicJRadioButton; // bold and italic
private ButtonGroup radioGroup; // buttongroup to hold radio buttons

// RadioButtonFrame constructor adds JRadioButtons to JFrame
public RadioButtonFrame()
 {
super( "RadioButton Test" );
setLayout( new FlowLayout() ); // set frame layout
textField = new JTextField( "Watch the font style change", 25 );
add( textField ); // add textField to JFrame

 // create radio buttons
plainJRadioButton = new JRadioButton( "Plain", true );
boldJRadioButton = new JRadioButton( "Bold", false );
italicJRadioButton = new JRadioButton( "Italic", false );
boldItalicJRadioButton = new JRadioButton( "Bold/Italic", false );
add( plainJRadioButton ); // add plain button to JFrame
add( boldJRadioButton ); // add bold button to JFrame
add( italicJRadioButton ); // add italic button to JFrame
add( boldItalicJRadioButton ); // add bold and italic button

// create logical relationship between JRadioButtons
radioGroup = new ButtonGroup(); // create ButtonGroup
radioGroup.add( plainJRadioButton ); // add plain to group
radioGroup.add( boldJRadioButton ); // add bold to group
radioGroup.add( italicJRadioButton ); // add italic to group
radioGroup.add( boldItalicJRadioButton ); // add bold and italic

// create font objects
plainFont = new Font( "Serif", Font.PLAIN, 14 );
boldFont = new Font( "Serif", Font.BOLD, 14 );
italicFont = new Font( "Serif", Font.ITALIC, 14 );
boldItalicFont = new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );
textField.setFont( plainFont ); // set initial font to plain

// register events for JRadioButtons
plainJRadioButton.addItemListener(
new RadioButtonHandler( plainFont ) );
boldJRadioButton.addItemListener(
new RadioButtonHandler( boldFont ) );
```

```
italicJRadioButton.addItemListener(
new RadioButtonHandler( italicFont ) );
boldItalicJRadioButton.addItemListener(
new RadioButtonHandler( boldItalicFont ) );
} // end RadioButtonFrame constructor

// private inner class to handle radio button events
private class RadioButtonHandler implements ItemListener
{
private Font font; // font associated with this listener
public RadioButtonHandler( Font f )
{
font = f; // set the font of this listener
 } // end constructor RadioButtonHandler

 // handle radio button events
public void itemStateChanged( ItemEvent event )
{
textField.setFont( font ); // set font of textField
} // end method itemStateChanged
} // end private inner class RadioButtonHandler
} // end class RadioButtonFrame


//RadioButtonTest.java
// Testing RadioButtonFrame.
import javax.swing.JFrame;

public class RadioButtonTest
{
public static void main( String[] args )
{
RadioButtonFrame radioButtonFrame = new RadioButtonFrame();
radioButtonFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
radioButtonFrame.setSize( 300, 100 ); // set frame size
radioButtonFrame.setVisible( true ); // display frame
} // end main
} // end class RadioButtonTest
```

output

JComboBox (Using an Anonymous Inner Class for Event Handling)

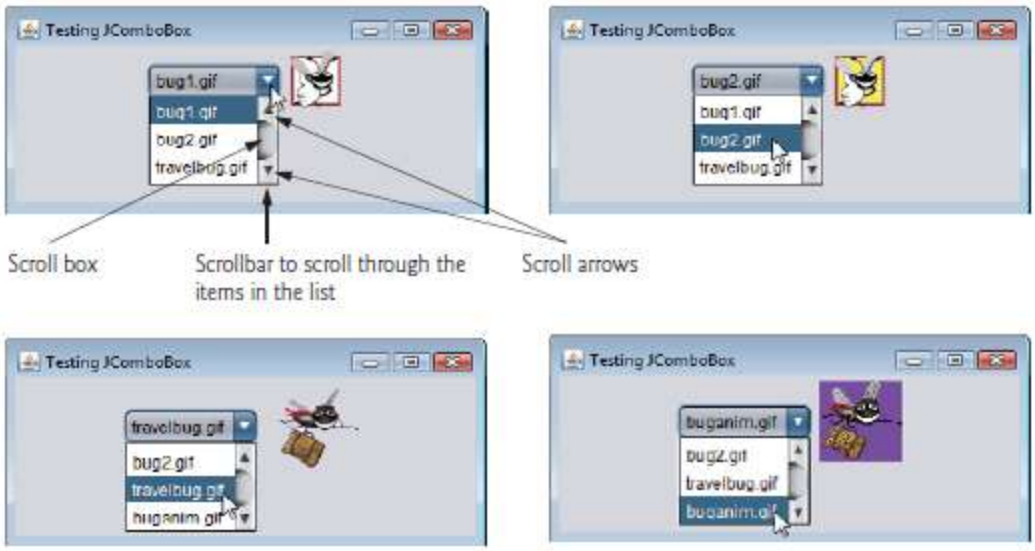
A combo box (sometimes called a **drop-down list**) **enables the user to select one item from a list** . Combo boxes are implemented with class **JComboBox**, which extends class JComponent. JComboBoxes generate ItemEvents just as JCheckBoxes and JRadioButtons do. This example also demonstrates a special form of inner class that's used frequently in event handling.

```
// Fig. 14.21: ComboBoxFrame.java
// JComboBox that displays a list of image names.
import java.awt.FlowLayout;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JComboBox;
import javax.swing.Icon;
import javax.swing.ImageIcon;

public class ComboBoxFrame extends JFrame
{
private JComboBox imagesJComboBox; // combobox to hold names of icons
private JLabel label; // label to display selected icon

private static final String[] names =
{ "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };
private Icon[] icons = {
new ImageIcon( getClass().getResource( names[ 0 ] ) ),
new ImageIcon( getClass().getResource( names[ 1 ] ) ),
new ImageIcon( getClass().getResource( names[ 2 ] ) ),
new ImageIcon( getClass().getResource( names[ 3 ] ) ) };

// ComboBoxFrame constructor adds JComboBox to JFrame
public ComboBoxFrame()
{
super( "Testing JComboBox" );
setLayout( new FlowLayout() ); // set frame layout
imagesJComboBox = new JComboBox( names ); // set up JComboBox
imagesJComboBox.setMaximumRowCount( 3 ); // display three rows
imagesJComboBox.addItemListener(
new ItemListener() // anonymous inner class
{
```

```
// handle JComboBox event
public void itemStateChanged( ItemEvent event )
{
// determine whether item selected
if ( event.getStateChange() == ItemEvent.SELECTED )
label.setIcon( icons[
imagesJComboBox.getSelectedIndex() ] );
} // end method itemStateChanged
} // end anonymous inner class
); // end call to addItemListener
add( imagesJComboBox ); // add combobox to JFrame
label = new JLabel( icons[ 0 ] ); // display first icon
add( label ); // add label to JFrame
} // end ComboBoxFrame constructor
}
```

```
// Fig. 14.22: ComboBoxTest.java
// Testing ComboBoxFrame.
import javax.swing.JFrame;

public class ComboBoxTest
{
public static void main( String[] args )
{
ComboBoxFrame comboBoxFrame = new ComboBoxFrame();
comboBoxFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
comboBoxFrame.setSize( 350, 150 ); // set frame size
comboBoxFrame.setVisible( true ); // display frame
} // end main
} // end class ComboBoxTest
```

Output



JList

A list displays a series of items from which the user may select one or more items . Lists are created with class **JList**, which directly extends class JComponent. Class JList supports **single selection lists (which allow only one item to be selected at a time)** and **multiple-selection lists(which allow any number of items to be selected)**.

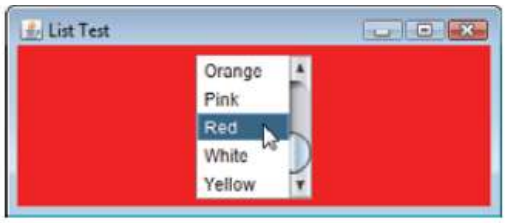
single-selection lists

The application below creates a JList containing 13 color names.When a color name is clicked in the JList, a **ListSelectionEvent** occurs and the application changes the background color of the application window to the selected color.

```
// ListFrame.java
// JList that displays a list of colors.
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionListener;
import javax.swing.event.ListSelectionEvent;
import javax.swing.ListSelectionModel;

public class ListFrame extends JFrame
 {
private JList colorJList; // list to display colors
private static final String[] colorNames = { "Black", "Blue", "Cyan",
"Dark Gray", "Gray", "Green", "Light Gray", "Magenta",
"Orange", "Pink", "Red", "White", "Yellow" };
private static final Color[] colors = { Color.BLACK, Color.BLUE,
Color.CYAN, Color.DARK_GRAY, Color.GRAY, Color.GREEN,
```

```
Color.LIGHT_GRAY, Color.MAGENTA, Color.ORANGE, Color.PINK,
Color.RED, Color.WHITE, Color.YELLOW };
// ListFrame constructor add JScrollPane containing JList to JFrame
public ListFrame()
{
super( "List Test" );
setLayout( new FlowLayout() ); // set frame layout
colorJList = new JList( colorNames ); // create with colorNames
colorJList.setVisibleRowCount( 5 ); // display five rows at once
// do not allow multiple selections
colorJList.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );
// add a JScrollPane containing JList to frame
add( new JScrollPane( colorJList ) );

colorJList.addListSelectionListener(
new ListSelectionListener() // anonymous inner class
{
// handle list selection events
public void valueChanged( ListSelectionEvent event )
{
getContentPane().setBackground(
colors[colorJList.getSelectedIndex() ] );
} // end method valueChanged
} // end anonymous inner class
); // end call to addListSelectionListener
} // end ListFrame constructor
} // end class ListFrame

// ListTest.java
//Selecting colors from a JList.
import javax.swing.JFrame;

public class ListTest
{
public static void main( String[] args )
{
ListFrame listFrame = new ListFrame(); // create ListFrame
listFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
listFrame.setSize( 350, 150 ); // set frame size
listFrame.setVisible( true ); // display frame
} // end main
} // end class ListTest
```

output



Multiple-Selection Lists

A multiple-selection list **enables the user to select many items from a JList.**

```
// MultipleSelectionFrame.java
//Copying items from one List to another.
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

public class MultipleSelectionFrame extends JFrame
 {
private JList colorJList; // list to hold color names
private JList copyJList; // list to copy color names into
private JButton copyJButton; // button to copy selected names
private static final String[] colorNames = { "Black", "Blue", "Cyan",
"Dark Gray", "Gray", "Green", "Light Gray", "Magenta", "Orange",
"Pink", "Red", "White", "Yellow" };

// MultipleSelectionFrame constructor
public MultipleSelectionFrame()
{
super( "Multiple Selection Lists" );
setLayout( new FlowLayout() ); // set frame layout

colorJList = new JList( colorNames ); // holds names of all colors
colorJList.setVisibleRowCount( 5 ); // show five rows
colorJList.setSelectionMode(
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
add( new JScrollPane( colorJList ) ); // add list with scrollpane

copyJButton = new JButton( "Copy >>>" ); // create copy button
copyJButton.addActionListener(

new ActionListener() // anonymous inner class
 {
```

```
// handle button event
public void actionPerformed( ActionEvent event )
{
// place selected values in copyJList
copyJList.setListData( colorJList.getSelectedValues() );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener

add( copyJButton ); // add copy button to JFrame

copyJList = new JList(); // create list to hold copied color names
copyJList.setVisibleRowCount( 5 ); // show 5 rows
copyJList.setFixedCellWidth( 100 ); // set width
copyJList.setFixedCellHeight( 15 ); // set height
copyJList.setSelectionMode(
ListSelectionModel.SINGLE_INTERVAL_SELECTION );
add( new JScrollPane( copyJList ) ); // add list with scrollpane
} // end MultipleSelectionFrame constructor
} // end class MultipleSelectionFrame


//MultipleSelectionTest.java
//Testing MultipleSelectionFrame.
import javax.swing.JFrame;

public class MultipleSelectionTest
{
public static void main( String[] args )
{
MultipleSelectionFrame multipleSelectionFrame =
new MultipleSelectionFrame();
multipleSelectionFrame.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE );
multipleSelectionFrame.setSize( 350, 150 ); // set frame size
multipleSelectionFrame.setVisible( true ); // display frame
} // end main
} // end class MultipleSelectionTest
```

output

Mouse Event Handling

MouseListener and MouseMotionListener interface methods

Methods of interface `MouseListener`

`public void mousePressed( MouseEvent event )`

Called when a mouse button is *pressed* while the mouse cursor is on a component.

`public void mouseClicked( MouseEvent event )`

Called when a mouse button is *pressed and released* while the mouse cursor remains stationary on a component. This event is always preceded by a call to `mousePressed`.

`public void mouseReleased( MouseEvent event )`

Called when a mouse button is *released after being pressed*. This event is always preceded by a call to `mousePressed` and one or more calls to `mouseDragged`.

`public void mouseEntered( MouseEvent event )`

Called when the mouse cursor *enters* the bounds of a component.

`public void mouseExited( MouseEvent event )`

Called when the mouse cursor *leaves* the bounds of a component.

Methods of interface `MouseMotionListener`

`public void mouseDragged( MouseEvent event )`

Called when the mouse button is *pressed* while the mouse cursor is on a component and the mouse is *moved* while the mouse button *remains pressed*. This event is always preceded by a call to `mousePressed`. All drag events are sent to the component on which the user began to drag the mouse.

`public void mouseMoved( MouseEvent event )`

Called when the mouse is *moved* (with no mouse buttons pressed) when the mouse cursor is on a component. All move events are sent to the component over which the mouse is currently positioned.

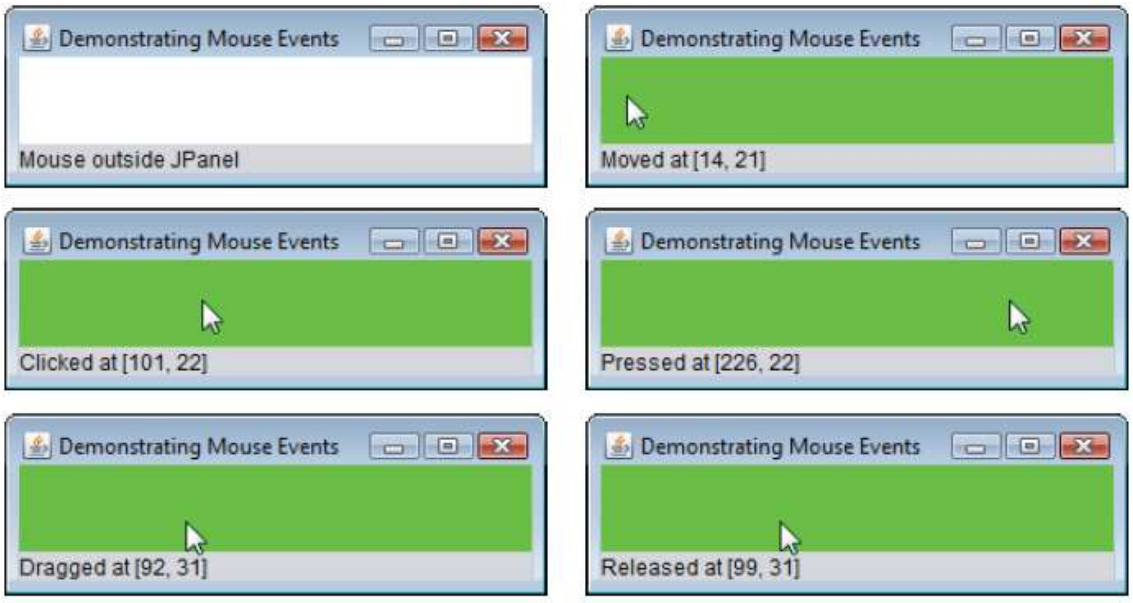
```
//MouseTrackerFrame.java
//Demonstrating mouse events.
import java.awt.Color;
import java.awt.BorderLayout;
```

```
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class MouseTrackerFrame extends JFrame
{
private JPanel mousePanel; // panel in which mouse events will occur
private JLabel statusBar; // label that displays event information

// MouseTrackerFrame constructor sets up GUI and
// registers mouse event handlers
public MouseTrackerFrame()
{
super( "Demonstrating Mouse Events" );
mousePanel = new JPanel(); // create panel
mousePanel.setBackground( Color.WHITE ); // set background color
add( mousePanel, BorderLayout.CENTER ); // add panel to JFrame
statusBar = new JLabel( "Mouse outside JPanel" );
add( statusBar, BorderLayout.SOUTH ); // add label to JFrame
// create and register listener for mouse and mouse motion events
MouseHandler handler = new MouseHandler();
mousePanel.addMouseListener( handler );
mousePanel.addMouseMotionListener( handler );
} // end MouseTrackerFrame constructor
private class MouseHandler implements MouseListener,
MouseMotionListener
{
 // MouseListener event handlers
 // handle event when mouse released immediately after press
public void mouseClicked( MouseEvent event )
{
statusBar.setText( String.format( "Clicked at [%d, %d]",
event.getX() event.getY() ));
} // end method mouseClicked

// handle event when mouse pressed

public void mousePressed( MouseEvent event )
statusBar.setText( String.format( "Pressed at [%d, %d]",
event.getX() event.getY() ));
} // end method mousePressed

// handle event when mouse released
public void mouseReleased( MouseEvent event )
{
```

```
statusBar.setText( String.format( "Released at [%d, %d]",
event.getX() event.getY() ));
 } // end method mouseReleased

// handle event when mouse enters area
public void mouseEntered( MouseEvent event )
statusBar.setText( String.format( "Mouse entered at [%d, %d]",
event.getX() ,event.getY() ));
mousePanel.setBackground( Color.GREEN );
} // end method mouseEntered

// handle event when mouse exits area
public void mouseExited( MouseEvent event )
{
statusBar.setText( "Mouse outside JPanel" );
mousePanel.setBackground( Color.WHITE );
} // end method mouseExited

// MouseMotionListener event handlers
// handle event when user drags mouse with button pressed
public void mouseDragged( MouseEvent event )
{
statusBar.setText( String.format( "Dragged at [%d, %d]",
event.getX() event.getY()));
} // end method mouseDragged

// handle event when user moves mouse
public void mouseMoved( MouseEvent event )
{
statusBar.setText( String.format( "Moved at [%d, %d]",
event.getX() ,event.getY() ));
} // end method mouseMoved
} // end inner class MouseHandler
} // end class MouseTrackerFrame

// MouseTrackerFrame.java
// Testing MouseTrackerFrame.
import javax.swing.JFrame;

public class MouseTracker
{
public static void main( String[] args )
{
MouseTrackerFrame mouseTrackerFrame = new MouseTrackerFrame();
mouseTrackerFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
mouseTrackerFrame.setSize( 300, 100 ); // set frame size
mouseTrackerFrame.setVisible( true ); // display frame
} // end main
```

} // end class MouseTracker

output



Adapter Classes

Many event-listener interfaces, such as MouseListener and MouseMotionListener, contain multiple methods. It's not always desirable to declare every method in an event-listener interface. For instance, an application may need only the mouseClicked handler from MouseListener or the mouseDragged handler from MouseMotionListener. Interface WindowListener specifies seven window event-handling methods. For many of the listener interfaces that have multiple methods, packages java.awt.event and javax.swing.event provide **event-listener adapter classes**. An adapter class implements an interface and provides a default implementation (with an empty method body) of each method in the interface. Figure below shows several java.awt.event adapter classes and the interfaces they implement. You can extend an adapter class to inherit the default implementation of every method and subsequently override only the method(s) you need for event handling.

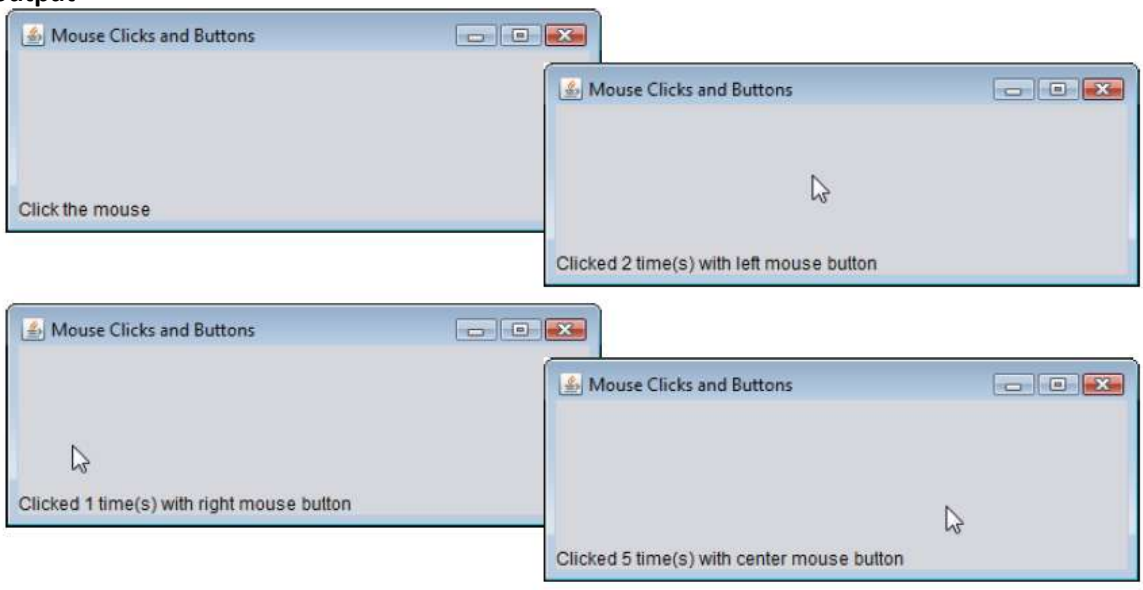
Event-adapter class in java.awt.event	Implements interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

fig. Event-adapter classes and the interfaces they implement in package java.awt.event.

```
//MouseDetailsFrame.java
// Demonstrating mouse clicks and distinguishing between mouse buttons.
import java.awt.BorderLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MouseDetailsFrame extends JFrame
 {
private String details; // String that is displayed in the statusBar
private JLabel statusBar; // JLabel that appears at bottom of window

// constructor sets title bar String and register mouse listener
public MouseDetailsFrame()
{
super( "Mouse clicks and buttons" );

statusBar = new JLabel( "Click the mouse" );
add( statusBar, BorderLayout.SOUTH );
addMouseListener( new MouseClickHandler() ); // add handler
} // end MouseDetailsFrame constructor

// inner class to handle mouse events
private class MouseClickHandler extends MouseAdapter
 {
// handle mouse-click event and determine which button was pressed
public void mouseClicked( MouseEvent event )
{
int xPos = event.getX(); // get x-position of mouse
int yPos = event.getY(); // get y-position of mouse
```

```
details = String.format( "Clicked %d time(s)",
event.getClickCount() );
if ( event.isMetaDown());
details += " with right mouse button";
else if( event.isAltDown())// middle mouse button
details += " with center mouse button";
else // left mouse button
details += " with left mouse button";

statusBar.setText( details ); // display message in statusBar
} // end method mouseClicked
} // end private inner class MouseClickHandler
} // end class MouseDetailsFrame

//MouseDetails.java
// Testing MouseDetailsFrame.
import javax.swing.JFrame;
public class MouseDetails
{
public static void main( String[] args )
{
MouseDetailsFrame mouseDetailsFrame = new MouseDetailsFrame();
mouseDetailsFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
mouseDetailsFrame.setSize( 400, 150 ); // set frame size
mouseDetailsFrame.setVisible( true ); // display frame
} // end main
} // end class MouseDetails
```

output



InputEvent method	Description
`isMetaDown()`	Returns `true` when the user clicks the *right mouse button* on a mouse with two or three buttons. To simulate a right-mouse-button click on a one-button mouse, the user can hold down the *Meta* key on the keyboard and click the mouse button.
`isAltDown()`	Returns `true` when the user clicks the *middle mouse button* on a mouse with three buttons. To simulate a middle-mouse-button click on a one- or two-button mouse, the user can press the *Alt* key and click the only or left mouse button, respectively.

Key Event Handling with JTextArea

Key events are generated when keys on the keyboard are pressed and released. A class that implements KeyListener must provide declarations for methods **keyPressed, keyReleased and key-Typed,** each of which receives a KeyEvent as its argument. Class KeyEvent is a subclass of InputEvent. **Method keyPressed** is called in response to **pressing any key**. **Method key- Typed** is called in response to pressing any key that is **not an action key**. (The action keys are any arrow key, Home, End, Page Up, Page Down, any function key, etc.) **Method key- Released** is called **when the key is released after any keyPressed or keyTyped event.**

The application below demonstrates the KeyListener methods. Class KeyDemoFrame implements the KeyListener interface, **so all three methods are declared in the application.** The constructor registers the application to handle its own key events by using method addKeyListener . Method addKey- Listener is declared in class Component, so every subclass of Component can notify Key- Listener objects of key events for that Component.

setDisabledTextColor to change the text color in the JTextArea to black for readability.
getKeyCode to get the virtual key code of the pressed key.Class KeyEvent contains virtual key-code constants that represent every key on the keyboard.
getKey-Text the value returned by getKeyCode is passed to static KeyEvent method getKey- Text, which returns a string containing the name of the key that was pressed.
get-KeyChar (which returns a char) to get the Unicode value of the character typed.
isActionKey to determine whether the key in the event was an action key.
getModifiers is called to determine whether any **modifier keys (such as Shift, Alt and Ctrl)** were pressed when the key event occurred.
getKeyModifiersText which produces a string containing the names of the pressed modifier keys.

```
//KeyDemoFrame.java
//Demonstrating keystroke events.
import java.awt.Color;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import javax.swing.JFrame;
import javax.swing.JTextArea;
```

```
public class KeyDemoFrame extends JFrame implements KeyListener
{
private String line1 = ""; // first line of textarea
private String line2 = ""; // second line of textarea
private String line3 = ""; // third line of textarea
private JTextArea textArea; // textarea to display output
// KeyDemoFrame constructor
public KeyDemoFrame()
{
super( "Demonstrating Keystroke Events" );

textArea = new JTextArea( 10, 15 ); // set up JTextArea
textArea.setText( "Press any key on the keyboard..." );
textArea.setEnabled( false ); // disable textarea
textArea.setDisabledTextColor( Color.BLACK ); // set text color
add( textArea ); // add textarea to JFrame
addKeyListener( this ); // allow frame to process key events
} // end KeyDemoFrame constructor

// handle press of any key
public void keyPressed( KeyEvent event )
{
line1 = String.format( "Key pressed: %s",KeyEvent.getKeyText( event.getKeyCode() ) ); // show pressed
key
setLines2and3( event ); // set output lines two and three
 } // end method keyPressed

// handle release of any key
public void keyReleased( KeyEvent event )
{
line1 = String.format( "Key released: %s",KeyEvent.getKeyText( event.getKeyCode() ) ); // show released
key
setLines2and3( event ); // set output lines two and three
} // end method keyReleased

// handle press of an action key
public void keyTyped( KeyEvent event )
{
line1 = String.format( "Key typed: %s",event.getKeyChar() );
setLines2and3( event ); // set output lines two and three
} // end method keyTyped

 // set second and third lines of output
private void setLines2and3( KeyEvent event )
{
line2 = String.format( "This key is %san action key",
(event.isActionKey() ?"" : "not " ) );
String temp = KeyEvent.getKeyModifiersText( event.getModifiers() );
```

```
line3 = String.format( "Modifier keys pressed: %s",
( temp.equals( "" ) ? "none" : temp ) ); // output modifiers

 textArea.setText( String.format( "%s\n%s\n%s\n",
line1, line2, line3 ) ); // output three lines of text
} // end method setLines2and3
} // end class KeyDemoFrame


//KeyDemo.java
// Testing KeyDemoFrame.
import javax.swing.JFrame;
public class KeyDemo
{
public static void main( String[] args )
{
KeyDemoFrame keyDemoFrame = new KeyDemoFrame();
keyDemoFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
keyDemoFrame.setSize( 350, 100 ); // set frame size
keyDemoFrame.setVisible( true ); // display frame
} // end main
} // end class KeyDemo
```

output

Introduction to Layout Managers

Layout managers arrange GUI components in a container for presentation purposes. You can use the layout managers for basic layout capabilities instead of determining every GUI component's exact

position and size. This functionality enables you to concentrate on the basic look-and-feel and lets the layout managers process most of the layout details. All layout managers implement the interface **LayoutManager** (in package java.awt). Class Container's **setLayout** method takes an object that implements the LayoutManager interface as an argument. There are basically three ways for you to arrange components in a GUI:

1. **Absolute positioning:** This provides the greatest level of control over a GUI's appearance. By setting a Container's layout to null, you can specify the absolute position of each GUI component with respect to the upper-left corner of the Container by using Component methods setSize and setLocation or setBounds. If you do this, you also must specify each GUI component's size. Programming a GUI with absolute positioning can be tedious, unless you have an integrated development environment (IDE) that can generate the code for you.

2. **Layout managers**: Using layout managers to position elements can be simpler and faster than creating a GUI with absolute positioning, but you lose some control over the size and the precise positioning of GUI components.

3. **Visual programming in an IDE:** IDEs provide tools that make it easy to create GUIs. Each IDE typically provides a GUI design tool that allows you to drag and drop GUI components from a tool box onto a design area. You can then position, size and align GUI components as you like. The IDE generates the Java code that creates the GUI. In addition, you can typically add event-handling code for a particular component by double-clicking the component.

Layout manager	Description
`FlowLayout`	Default for `javax.swing.JPanel`. Places components sequentially (left to right) in the order they were added. It's also possible to specify the order of the components by using the `Container` method `add`, which takes a `Component` and an integer index position as arguments.
`BorderLayout`	Default for `JFrames` (and other windows). Arranges the components into five areas: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
`GridLayout`	Arranges the components into rows and columns.

FlowLayout

FlowLayout is the simplest layout manager. **GUI components are placed on a container from left to right in the order in which they're added to the container**. When the edge of the container is reached, components continue to display on the next line.

Class FlowLayout allows GUI components to be **left aligned, centered (the default) and right aligned.**

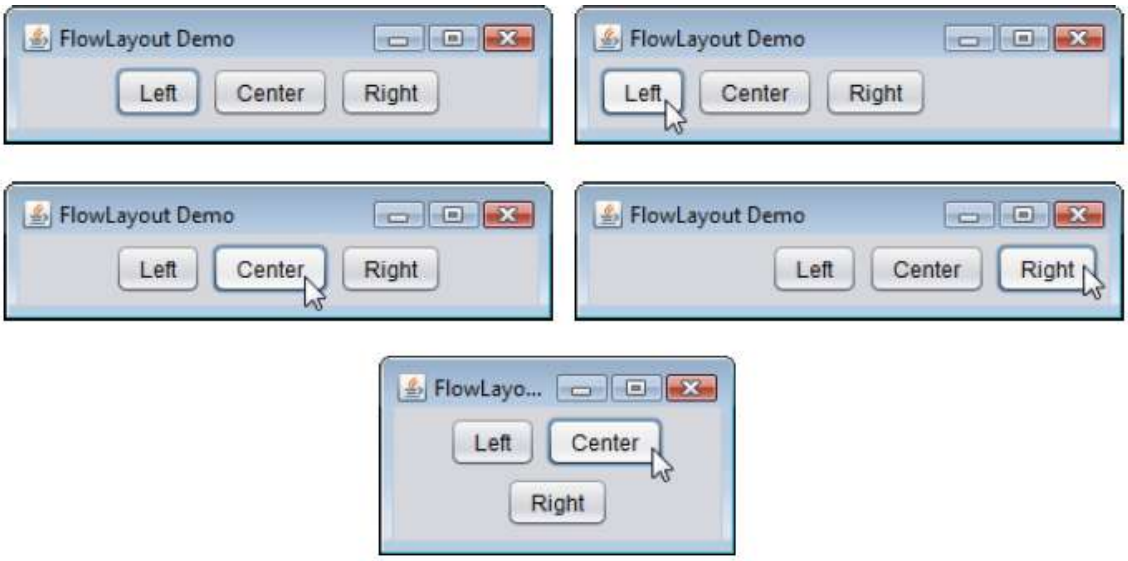
The application below creates three JButton objects and adds them to the application, using a FlowLayout layout manager. The components are center aligned by default. When the user clicks Left, the alignment for the layout manager is changed to a left-aligned FlowLayout. When the user clicks Right, the alignment for the layout manager is changed to a right-aligned FlowLayout. When the user clicks Center, the alignment for the layout manager is changed to a center-aligned FlowLayout.

```
//FlowLayoutFrame.java
//Demonstrating FlowLayout alignments.
```

```
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class FlowLayoutFrame extends JFrame
{
private JButton leftJButton; // button to set alignment left
private JButton centerJButton; // button to set alignment center
private JButton rightJButton; // button to set alignment right
private FlowLayout layout; // layout object
private Container container; // container to set layout

 // set up GUI and register button listeners
public FlowLayoutFrame()
{
super( "FlowLayout Demo" );
layout = new FlowLayout(); // create FlowLayout
container = getContentPane(); // get container to layout
setLayout( layout ); // set frame layout
// set up leftJButton and register listener
leftJButton = new JButton( "Left" ); // create Left button
add( leftJButton ); // add Left button to frame
leftJButton.addActionListener(
new ActionListener() // anonymous inner class
 {
// process leftJButton event
public void actionPerformed( ActionEvent event )
{
layout.setAlignment( FlowLayout.LEFT );
// realign attached components
layout.layoutContainer( container );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener

// set up centerJButton and register listener
centerJButton = new JButton( "Center" ); // create Center button
add( centerJButton ); // add Center button to frame
centerJButton.addActionListener(
new ActionListener() // anonymous inner class
{
// process centerJButton event
public void actionPerformed( ActionEvent event )
{
layout.setAlignment( FlowLayout.CENTER );
```

```
// realign attached components
layout.layoutContainer( container );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener

// set up rightJButton and register listener
rightJButton = new JButton( "Right" ); // create Right button
add( rightJButton ); // add Right button to frame
rightJButton.addActionListener(
new ActionListener() // anonymous inner class
{
// process rightJButton event
public void actionPerformed( ActionEvent event )
{
layout.setAlignment( FlowLayout.RIGHT );
// realign attached components
layout.layoutContainer( container );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener
} // end FlowLayoutFrame constructor
} // end class FlowLayoutFrame


// FlowLayoutDemo.java
// Testing FlowLayoutFrame.
import javax.swing.JFrame;

public class FlowLayoutDemo
{
public static void main( String[] args )
{
FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
flowLayoutFrame.setSize( 300, 75 ); // set frame size
flowLayoutFrame.setVisible( true ); // display frame
} // end main
} // end class FlowLayoutDemo
```

output



BorderLayout

The **BorderLayout layout manager (the default layout manager for a JFrame)** arranges components into **five regions: NORTH, SOUTH, EAST, WEST and CENTER. NORTH corresponds to the top of the container.**

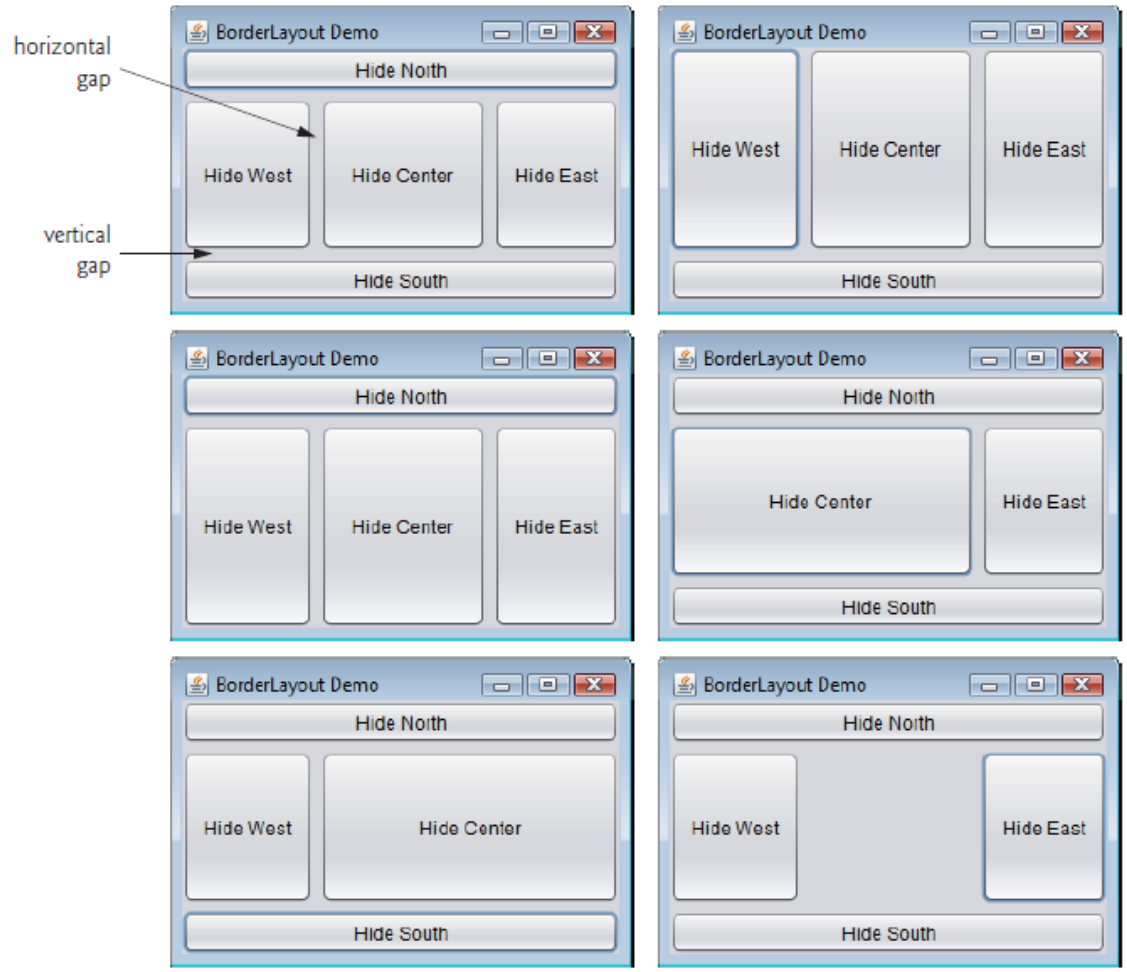
A BorderLayout limits a Container to containing at most five components—one in each region. The components placed in the **NORTH and SOUTH regions extend horizontally** to the sides of the container and are as tall as the components placed in those regions. The **EAST and WEST regions expand vertically between the NORTH and SOUTH** regions and are as wide as the components placed in those regions. The component placed in the **CENTER region expands to fill all remaining space in the layout** .

If all five regions are occupied, the entire container's space is covered by GUI components. If the NORTH or SOUTH region is not occupied, the GUI components in the EAST, CENTER and WEST regions expand vertically to fill the remaining space. If the EAST or WEST region is not occupied, the GUI component in the CENTER region expands horizontally to fill the remaining space. If the CENTER region is not occupied, the area is left empty—the other GUI components do not expand

```
// BorderLayoutFrame.java
// Demonstrating BorderLayout.
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class BorderLayoutFrame extends JFrame implements ActionListener
{
private JButton[] buttons; // array of buttons to hide portions
private static final String[] names = { "Hide North", "Hide South",
"Hide East", "Hide West", "Hide Center" };
private BorderLayout layout; // borderlayout object
// set up GUI and event handling
public BorderLayoutFrame()
```

```
{
super( "BorderLayout Demo" );
layout = new BorderLayout( 5, 5 ); // 5 pixel gaps
setLayout( layout ); // set frame layout
buttons = new JButton[ names.length ]; // set size of array

 // create JButtons and register listeners for them
for ( int count = 0; count < names.length; count++ )
{
buttons[ count ] = new JButton( names[ count ] );
buttons[ count ].addActionListener( this );
} // end for
add( buttons[ 0 ], BorderLayout.NORTH ); // add button to north
add( buttons[ 1 ], BorderLayout.SOUTH ); // add button to south
add( buttons[ 2 ], BorderLayout.EAST ); // add button to east
add( buttons[ 3 ], BorderLayout.WEST ); // add button to west
add( buttons[ 4 ], BorderLayout.CENTER ); // add button to center
} // end BorderLayoutFrame constructor

// handle button events
public void actionPerformed( ActionEvent event )
{
// check event source and lay out content pane correspondingly
for ( JButton button : buttons )
{
if ( event.getSource() == button )
button.setVisible( false ); // hide button clicked
else
button.setVisible( true ); // show other buttons
} // end for
layout.layoutContainer( getContentPane() ); // lay out content pane
} // end method actionPerformed
} // end class BorderLayoutFrame

// BorderLayoutDemo.java
// Testing BorderLayoutFrame.
import javax.swing.JFrame;

public class BorderLayoutDemo
{
public static void main( String[] args )
{
BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
borderLayoutFrame.setSize( 300, 200 ); // set frame size
borderLayoutFrame.setVisible( true ); // display frame
} // end main
} // end class BorderLayoutDemo
```

output



GridLayout

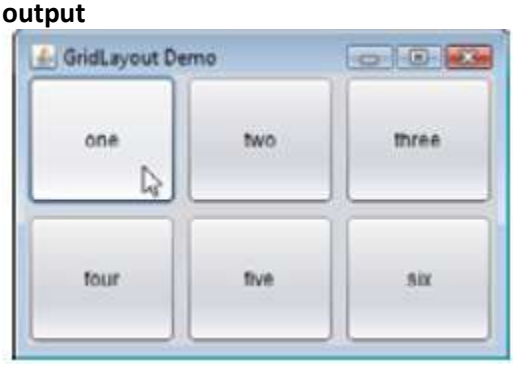
The **GridLayout layout manager** divides the container into a grid so that components can be placed in **rows and columns**. Class GridLayout inherits directly from class Object and implements interface LayoutManager. **Every Component in a GridLayout has the same width and height.** Components are added to a GridLayout starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid, and so on.

```
// GridLayoutFrame.java
// Demonstrating GridLayout.
import java.awt.GridLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class GridLayoutFrame extends JFrame implements ActionListener
{
```

```
private JButton[] buttons; // array of buttons
private static final String[] names =
{ "one", "two", "three", "four", "five", "six" };
private boolean toggle = true; // toggle between two layouts
private Container container; // frame container
private GridLayout gridLayout1; // first gridlayout
private GridLayout gridLayout2; // second gridlayout
// no-argument constructor
public GridLayoutFrame()
{
super( "GridLayout Demo" );
gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 by 3; gaps of 5
gridLayout2 = new GridLayout( 3, 2 ); // 3 by 2; no gap
container = getContentPane(); // get content pane
setLayout( gridLayout1 ); // set JFrame layout
buttons = new JButton[ names.length ]; // create array of JButtons

for ( int count = 0; count < names.length; count++ )
{

 buttons[ count ] = new JButton( names[ count ] );
buttons[ count ].addActionListener( this ); // register listener
add( buttons[ count ] ); // add button to JFrame
} // end for
} // end GridLayoutFrame constructor

 // handle button events by toggling between layouts
 public void actionPerformed( ActionEvent event )
{
if ( toggle )
container.setLayout( gridLayout2 ); // set layout to second
else
container.setLayout( gridLayout1 ); // set layout to first
toggle = !toggle; // set toggle to opposite value
container.validate(); // re-lay out container
} // end method actionPerformed
} // end class GridLayoutFrame


// GridLayoutDemo.java
// Testing GridLayoutFrame.
import javax.swing.JFrame;

public class GridLayoutDemo
{
public static void main( String[] args )
{
GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
```

```
gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
gridLayoutFrame.setSize( 300, 200 ); // set frame size
gridLayoutFrame.setVisible( true ); // display frame
} // end main
} // end class GridLayoutDemo
```

output

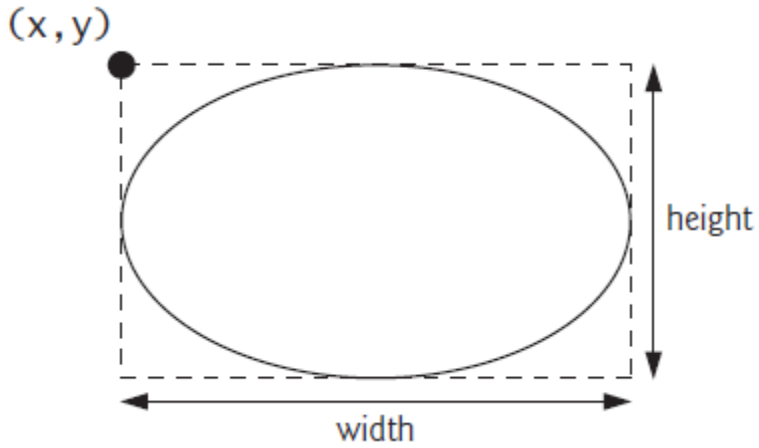


JSlider

JSliders enable a user to select from a range of **integer values**. Class JSlider inherits from JComponent. Figure below shows a horizontal JSlider with **tick marks** and the **thumb** that allows a user to select a value.

if a JSlider has the focus (i.e., it's the currently selected GUI component in the user interface), the **left arrow key and right arrow key cause the thumb of the JSlider to decrease or increase by 1, respectively**. **The down arrow key and up arrow key also cause the thumb to decrease or increase by 1 tick, respectively**. **The PgDn (page down) key and PgUp (page up) key cause the thumb to decrease or increase by block increments of one-tenth of the range of values, respectively**. **The Home key moves the thumb to the minimum value of the JSlider, and the End key moves the thumb to the maximum value of the JSlider.**

- **paintComponent** method that can be used to draw graphics.Method paintComponent takes a Graphics object as an argument. This object is passed to the paintComponent method by the system when a lightweight Swing component needs to be repainted.
- **repaint** method can be used if you want to update the graphics drawnon a Swing component.
- **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.



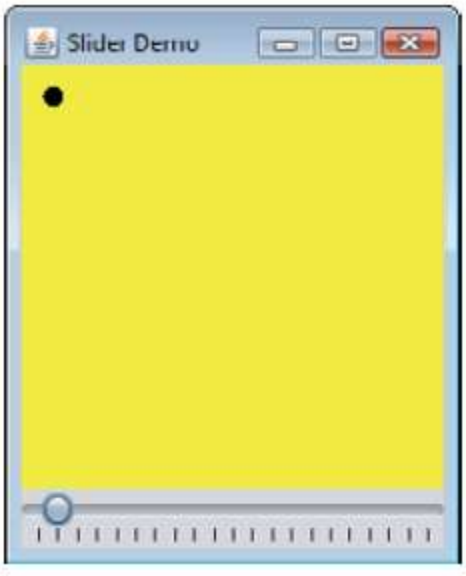
- The **JSlider constructor** takes four arguments. The first argument specifies the orientation of Slider, which is HORIZONTAL or VERTICAL (a constant in interface SwingConstants). The second and third arguments indicate the minimum and maximum integer values in the range of values for this JSlider. The last argument indicates that the initial value of the JSlider (i.e., where the thumb is displayed).
- Method **setMajorTickSpacing** indicates that each major tick mark represents 10 values in the range of values.
- Method **setPaintTicks** with a true argument indicates that the tick marks should be displayed (they aren't displayed by default).

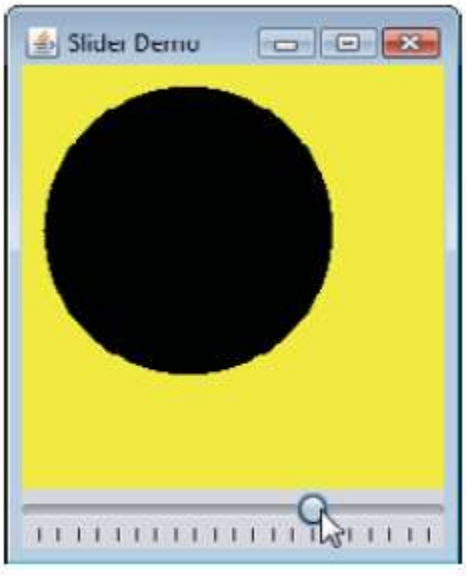
```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Dimension;
import javax.swing.JPanel;

public class OvalPanel extends JPanel
{
  private int diameter = 10; // default diameter of 10

  // draw an oval of the specified diameter
  public void paintComponent( Graphics g )
  {
    super.paintComponent( g );
     g.setColor(Color.red);
    g.fillOval( 10, 10, diameter, diameter ); // draw circle

  } // end method paintComponent

  // validate and set diameter, then repaint
  public void setDiameter( int newDiameter )
  {
    // if diameter invalid, default to 10
    diameter = ( newDiameter >= 0 ? newDiameter : 10 );
    repaint(); // repaint panel
```

```
  } // end method setDiameter

  // used by layout manager to determine preferred size
  public Dimension getPreferredSize()
  {
    return new Dimension( 200, 200 );
  } // end method getPreferredSize

  // used by layout manager to determine minimum size
  public Dimension getMinimumSize()
  {
    return getPreferredSize();
  } // end method getMinimumSize
} // end class OvalPanel

import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JSlider;
import javax.swing.SwingConstants;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

public class SliderFrame extends JFrame
{
  private JSlider diameterJSlider; // slider to select diameter
  private OvalPanel myPanel; // panel to draw circle

  // no-argument constructor
  public SliderFrame()
  {
    super( "Slider Demo" );

    myPanel = new OvalPanel(); // create panel to draw circle
    myPanel.setBackground( Color.YELLOW ); // set background to yellow

    // set up JSlider to control diameter value
    diameterJSlider =
      new JSlider( SwingConstants.HORIZONTAL, 0, 200, 10 );
    diameterJSlider.setMajorTickSpacing( 10 ); // create tick every 10
    diameterJSlider.setPaintTicks( true ); // paint ticks on slider

    // register JSlider event listener
    diameterJSlider.addChangeListener(

      new ChangeListener() // anonymous inner class
      {
        // handle change in slider value
```

```
        public void stateChanged( ChangeEvent e )
        {
          myPanel.setDiameter( diameterJSlider.getValue() );
        } // end method stateChanged
      } // end anonymous inner class
    ); // end call to addChangeListener

    add( diameterJSlider, BorderLayout.SOUTH ); // add slider to frame
    add( myPanel, BorderLayout.CENTER ); // add panel to frame
  } // end SliderFrame constructor
} // end class SliderFrame


import javax.swing.JFrame;

public class SliderDemo
{
  public static void main( String[] args )
  {
    SliderFrame sliderFrame = new SliderFrame();
    sliderFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    sliderFrame.setSize( 220, 270 ); // set frame size
    sliderFrame.setVisible( true ); // display frame
  } // end main
} // end class SliderDemo
```

Output





JColorChooser

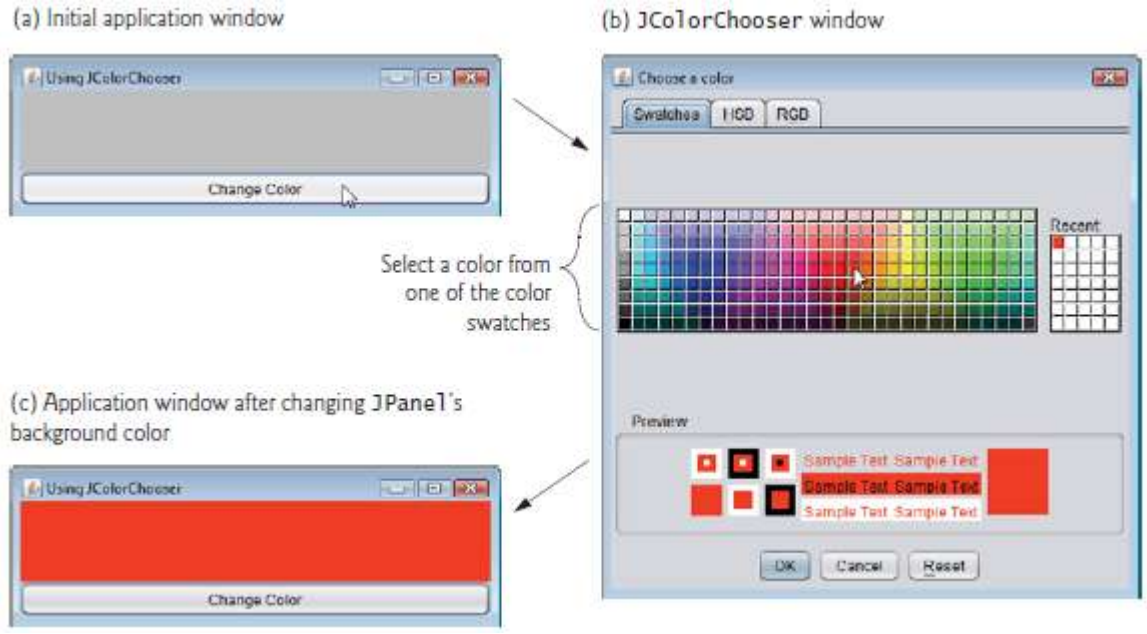
Package javax.swing provides the JColorChooser GUI component that enables application users to select colors. The application below demonstrates a JColorChooser dialog. When you click the Change Color

button, a JColorChooser dialog appears. When you select a color and press the dialog's OK button, the background color of the application window changes.

```
//ShowColors2JFrame.java
//Choosing colors with JColorChooser.
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JColorChooser;
import javax.swing.JPanel;

public class ShowColors2JFrame extends JFrame
{
private JButton changeColorJButton;
private Color color = Color.LIGHT_GRAY;
private JPanel colorJPanel;

// set up GUI
public ShowColors2JFrame()
{
super( "Using JColorChooser" );

// create JPanel for display color
colorJPanel = new JPanel();
colorJPanel.setBackground( color );

// set up changeColorJButton and register its event handler
changeColorJButton = new JButton( "Change Color" );
changeColorJButton.addActionListener(
new ActionListener() // anonymous inner class
{
// display JColorChooser when user clicks button
public void actionPerformed( ActionEvent event )
{
color = JColorChooser.showDialog(
ShowColors2JFrame.this, "Choose a color", color );
// set default color, if no color is returned
if ( color == null )
color = Color.LIGHT_GRAY;
// change content pane's background color
colorJPanel.setBackground( color );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener
```

```
add( colorJPanel, BorderLayout.CENTER ); // add colorJPanel
add( changeColorJButton, BorderLayout.SOUTH ); // add button

setSize( 400, 130 ); // set frame size
setVisible( true ); // display frame
} // end ShowColor2JFrame constructor
} // end class ShowColors2JFrame

//ShowColors2.java
// Choosing colors with JColorChooser.
import javax.swing.JFrame;

public class ShowColors2
{
// execute application
public static void main( String[] args )
{
ShowColors2JFrame application = new ShowColors2JFrame();
application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
} // end main
} // end class ShowColors2
```

output



Using Menus with Frames

Menus are an integral part of GUIs. They allow the user to perform actions without unnecessarily cluttering a GUI with extra components. In Swing GUIs, menus can be attached
only to objects of the classes that provide method setJMenuBar. Two such classes are **JFrame and JApplet**. The classes used to declare menus are **JMenuBar, JMenu, JMenu-Item, JCheckBoxMenuItem and class JRadioButtonMenuItem.**

Class JMenuBar (a subclass of JComponent) contains the methods necessary to manage a menu bar, which is a container for menus.

Class JMenu (a subclass of javax.swing.JMenu-Item) contains the methods necessary for managing menus. Menus contain **menu items** and are added to **menu bars** or to other menus as submenus. When a menu is clicked, it expands to show its list of menu items.

Class JMenuItem (a subclass of javax.swing.AbstractButton) contains the methods necessary to manage menu items. A **menu item** is a GUI component **inside a menu** that, when selected, causes an **action event**. A menu item can be used to initiate an action, or it can be a submenu that provides more menu items from which the user can select. Submenus are useful for grouping related menu items in a menu.

Class JCheckBoxMenuItem (a subclass of javax.swing.JMenuItem) contains the methods necessary to manage **menu items** that can be toggled on or off. When a JCheck-BoxMenuItem is selected, a check appears to the left of the menu item. When the JCheck-BoxMenuItem is selected again, the check is removed.

Class JRadioButtonMenuItem (a subclass of javax.swing.JMenuItem) contains the methods necessary to manage menu items that can be toggled on or off like JCheckBox-MenuItems. When multiple JRadioButtonMenuItems are maintained as part of a Button-Group, only one item in the group can be selected at a given time. When a JRadioButtonMenuItem is selected, a filled circle appears to the left of the menu item. When another JRadioButtonMenuItem is selected, the filled circle of the previously selected menu item is removed.

```
// MenuFrame.java
// Demonstrating menus.
import java.awt.Color;
import java.awt.Font;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.ButtonGroup;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;

public class MenuFrame extends JFrame
{
  private final Color[] colorValues =
    { Color.BLACK, Color.BLUE, Color.RED, Color.GREEN };
  private JRadioButtonMenuItem[] colorItems;    // color menu items
  private JRadioButtonMenuItem[] fonts;         // font menu items
  private JCheckBoxMenuItem[] styleItems;       // font style menu items
```

```
private JLabel displayJLabel;                    // displays sample text
private ButtonGroup fontButtonGroup;             // manages font menu items
private ButtonGroup colorButtonGroup;            // manages color menu items
private int style;                               // used to create style for font

// no-argument constructor set up GUI
public MenuFrame()
{
  super( "Using JMenus" );

  JMenu fileMenu = new JMenu( "File" );          // create file menu
  fileMenu.setMnemonic( 'F' );                   // set mnemonic to F

  // create About... menu item
  JMenuItem aboutItem = new JMenuItem( "About..." );
  aboutItem.setMnemonic( 'A' );                  // set mnemonic to A
  fileMenu.add( aboutItem );                     // add about item to file menu
  aboutItem.addActionListener(
    new ActionListener() // anonymous inner class
    {
      // display message dialog when user selects About...
      public void actionPerformed( ActionEvent event )
      {
        JOptionPane.showMessageDialog( MenuFrame.this,
          "This is an example\nof using menus",
          "About", JOptionPane.PLAIN_MESSAGE );
      }        // end method actionPerformed
    }          // end anonymous inner class
  );           // end call to addActionListener

  JMenuItem exitItem = new JMenuItem( "Exit" );    // create exit item
  exitItem.setMnemonic( 'x' );                     // set mnemonic to x
  fileMenu.add( exitItem );                        // add exit item to file menu
  exitItem.addActionListener(

    new ActionListener()                           // anonymous inner class
    {
      // terminate application when user clicks exitItem
      public void actionPerformed( ActionEvent event )
      {
        System.exit( 0 );        // exit application
      }                          // end method actionPerformed
    }                            // end anonymous inner class
  ); // end call to addActionListener

  JMenuBar bar = new JMenuBar();     // create menu bar
  setJMenuBar( bar );      // add menu bar to application
  bar.add( fileMenu );     // add file menu to menu bar
```

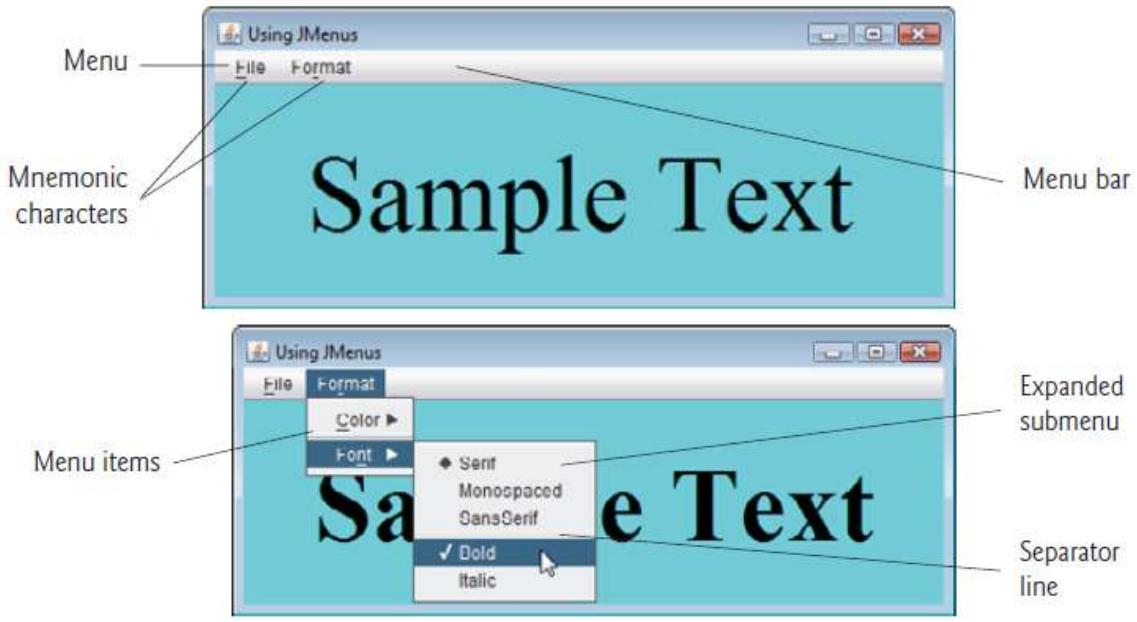
```
JMenu formatMenu = new JMenu( "Format" );    // create format menu
formatMenu.setMnemonic( 'r' );      // set mnemonic to r

// array listing string colors
String[] colors = { "Black", "Blue", "Red", "Green" };

JMenu colorMenu = new JMenu( "Color" );     // create color menu
colorMenu.setMnemonic( 'C' );     // set mnemonic to C

// create radio button menu items for colors
colorItems = new JRadioButtonMenuItem[ colors.length ];
colorButtonGroup = new ButtonGroup();      // manages colors
ItemHandler itemHandler = new ItemHandler();           // handler for colors

// create color radio button menu items
for ( int count = 0; count < colors.length; count++ )
{
  colorItems[ count ] =
    new JRadioButtonMenuItem( colors[ count ] );           // create item
  colorMenu.add( colorItems[ count ] );                // add item to color menu
  colorButtonGroup.add( colorItems[ count ] );               // add to group
  colorItems[ count ].addActionListener( itemHandler );
} // end for

colorItems[ 0 ].setSelected( true );    // select first Color item

formatMenu.add( colorMenu );         // add color menu to format menu
formatMenu.addSeparator();           // add separator in menu

// array listing font names
String[] fontNames = { "Serif", "Monospaced", "SansSerif" };
JMenu fontMenu = new JMenu( "Font" );      // create font menu
fontMenu.setMnemonic( 'n' );                       // set mnemonic to n

// create radio button menu items for font names
fonts = new JRadioButtonMenuItem[ fontNames.length ];
fontButtonGroup = new ButtonGroup(); // manages font names

// create Font radio button menu items
for ( int count = 0; count < fonts.length; count++ )
{
  fonts[ count ] = new JRadioButtonMenuItem( fontNames[ count ] );
  fontMenu.add( fonts[ count ] );     // add font to font menu
  fontButtonGroup.add( fonts[ count ] );   // add to button group
  fonts[ count ].addActionListener( itemHandler );   // add handler
} // end for
```

```
    fonts[ 0 ].setSelected( true );   // select first Font menu item
    fontMenu.addSeparator();      // add separator bar to font menu

    String[] styleNames = { "Bold", "Italic" };    // names of styles
    styleItems = new JCheckBoxMenuItem[ styleNames.length ];
    StyleHandler styleHandler = new StyleHandler();     // style handler

    // create style checkbox menu items
    for ( int count = 0; count < styleNames.length; count++ )
    {
      styleItems[ count ] =
        new JCheckBoxMenuItem( styleNames[ count ] ); // for style
      fontMenu.add( styleItems[ count ] ); // add to font menu
      styleItems[ count ].addItemListener( styleHandler ); // handler
    } // end for

    formatMenu.add( fontMenu ); // add Font menu to Format menu
    bar.add( formatMenu ); // add Format menu to menu bar

    // set up label to display text
    displayJLabel = new JLabel( "Sample Text", SwingConstants.CENTER );
    displayJLabel.setForeground( colorValues[ 0 ] );
    displayJLabel.setFont( new Font( "Serif", Font.PLAIN, 72 ) );

    getContentPane().setBackground( Color.CYAN ); // set background
    add( displayJLabel, BorderLayout.CENTER ); // add displayJLabel
  } // end MenuFrame constructor

  // inner class to handle action events from menu items
  private class ItemHandler implements ActionListener
  {
    // process color and font selections
    public void actionPerformed( ActionEvent event )
    {
      // process color selection
      for ( int count = 0; count < colorItems.length; count++ )
      {
        if ( colorItems[ count ].isSelected() )
        {
          displayJLabel.setForeground( colorValues[ count ] );
          break;
        } // end if
      } // end for

      // process font selection
      for ( int count = 0; count < fonts.length; count++ )
      {
        if ( event.getSource() == fonts[ count ] )
```

```
        {
          displayJLabel.setFont(
            new Font( fonts[ count ].getText(), style, 72 ) );
        } // end if
      } // end for

      repaint(); // redraw application
    } // end method actionPerformed
  } // end class ItemHandler

  // inner class to handle item events from checkbox menu items
  private class StyleHandler implements ItemListener
  {
    // process font style selections
    public void itemStateChanged( ItemEvent e )
    {
      String name = displayJLabel.getFont().getName(); // current Font
      Font font; // new font based on user selections

      // determine which CheckBoxes are checked and create Font
      if ( styleItems[ 0 ].isSelected() &&
          styleItems[ 1 ].isSelected() )
        font = new Font( name, Font.BOLD + Font.ITALIC, 72 );
      else if ( styleItems[ 0 ].isSelected() )
        font = new Font( name, Font.BOLD, 72 );
      else if ( styleItems[ 1 ].isSelected() )
        font = new Font( name, Font.ITALIC, 72 );
      else
        font = new Font( name, Font.PLAIN, 72 );

      displayJLabel.setFont( font );
      repaint(); // redraw application
    } // end method itemStateChanged
  } // end class StyleHandler
} // end class MenuFrame

//MenuTest.java
//Testing MenuFrame.
import javax.swing.JFrame;

public class PopupTest
{
  public static void main( String[] args )
  {
    PopupFrame popupFrame = new PopupFrame();    // create PopupFrame
    popupFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    popupFrame.setSize( 300, 200 );   // set frame size
    popupFrame.setVisible( true );   // display frame
```

```
  } // end main
} // end class PopupTest
```

output



JPopupMenu

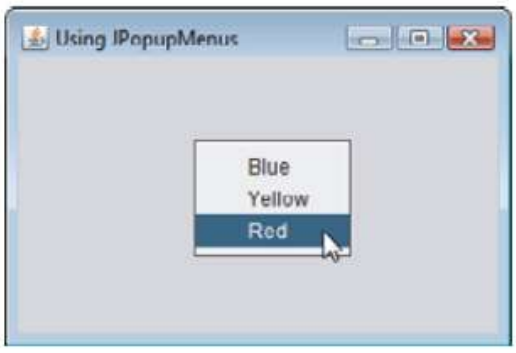
Many of today's computer applications provide so-called context-sensitive pop-up menus. In Swing, such menus are created with class **JPopupMenu** (a subclass of JComponent). These menus provide options that are specific to the component for which the popup trigger event was generated. On most systems, **the pop-up trigger event occurs when the user presses and releases the right mouse button.**

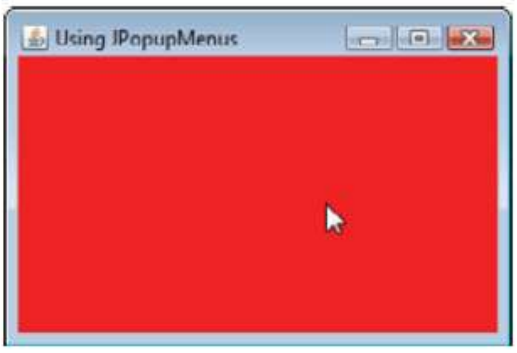
```
//PopupFrame.java
//Demonstrating JPopupMenus.
import java.awt.Color;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JRadioButtonMenuItem;
import javax.swing.JPopupMenu;
import javax.swing.ButtonGroup;

public class PopupFrame extends JFrame
{
private JRadioButtonMenuItem[] items; // holds items for colors
private final Color[] colorValues =
{ Color.BLUE, Color.YELLOW, Color.RED }; // colors to be used
private JPopupMenu popupMenu; // allows user to select color
// no-argument constructor sets up GUI
```

```
public PopupFrame()
{
 super( "Using JPopupMenus" );

ItemHandler handler = new ItemHandler(); // handler for menu items
String[] colors = { "Blue", "Yellow", "Red" }; // array of colors

ButtonGroup colorGroup = new ButtonGroup(); // manages color items
popupMenu = new JPopupMenu(); // create pop-up menu
items = new JRadioButtonMenuItem[ colors.length ]; // color items

// construct menu item, add to pop-up menu, enable event handling
for ( int count = 0; count < items.length; count++ )
 {
 items[ count ] = new JRadioButtonMenuItem( colors[ count ] );
popupMenu.add( items[ count ] ); // add item to pop-up menu
colorGroup.add( items[ count ] ); // add item to button group
items[ count ].addActionListener( handler ); // add handler
} // end for

setBackground( Color.WHITE ); // set background to white

// declare a MouseListener for the window to display pop-up menu
addMouseListener(

new MouseAdapter() // anonymous inner class
{
// handle mouse press event
public void mousePressed( MouseEvent event )
{
checkForTriggerEvent( event ); // check for trigger
} // end method mousePressed

// handle mouse release event
public void mouseReleased( MouseEvent event )
{
checkForTriggerEvent( event ); // check for trigger
} // end method mouseReleased

// determine whether event should trigger pop-up menu
private void checkForTriggerEvent( MouseEvent event )
{
if ( event.isPopupTrigger() )
popupMenu.show(
event.getComponent(), event.getX(), event.getY() );
} // end method checkForTriggerEvent
} // end anonymous inner class
); // end call to addMouseListener
```

```
} // end PopupFrame constructor

// private inner class to handle menu item events
private class ItemHandler implements ActionListener
 {
// process menu item selections
 public void actionPerformed( ActionEvent event )
 {
// determine which menu item was selected
for ( int i = 0; i < items.length; i++ )
 {
if ( event.getSource() == items[ i ] )
 {
getContentPane().setBackground( colorValues[ i ] );
return;
 } // end if
} // end for
} // end method actionPerformed
} // end private inner class ItemHandler
} // end class PopupFrame

//PopupTest.java
// Testing PopupFrame.
import javax.swing.JFrame;

public class PopupTest
{
public static void main( String[] args )
{
PopupFrame popupFrame = new PopupFrame(); // create PopupFrame
popupFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
popupFrame.setSize( 300, 200 ); // set frame size
popupFrame.setVisible( true ); // display frame
} // end main
 } // end class PopupTest
```

output





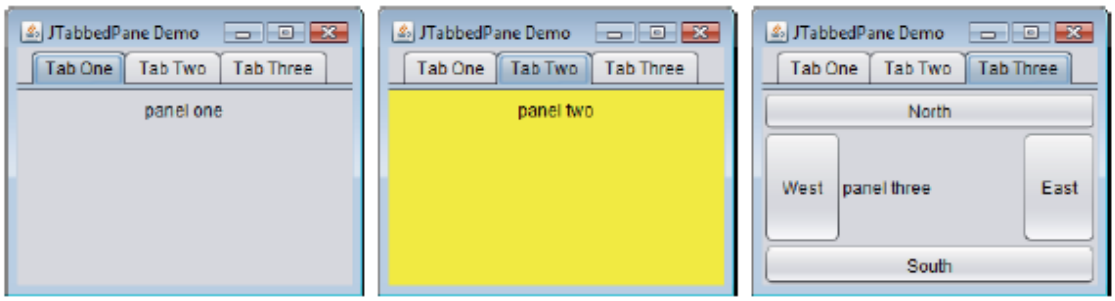
JTabbedPane

A **JTabbedPane** arranges GUI components into layers, of which only one is visible at a time. Users access each layer via a tab—similar to folders in a file cabinet. When the user clicks a tab, the appropriate layer is displayed. The tabs appear at the top by default but also can be positioned at the left, right or bottom of the JTabbedPane. Any component can be placed on a tab. If the component is a container, such as a panel, it can use any layout manager to lay out several components on the tab. Class JTabbedPane is a subclass of JComponent. The application below creates one tabbed pane with three tabs. Each tab displays one of the JPanels—panel1, panel2 or panel3.

```
// JTabbedPaneFrame.java
// Demonstrating JTabbedPane
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JTabbedPane;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.SwingConstants;

public class JTabbedPaneFrame extends JFrame
{
  // set up GUI
  public JTabbedPaneFrame()
  {
    super( "JTabbedPane Demo " );

    JTabbedPane tabbedPane = new JTabbedPane(); // create JTabbedPane

    // set up pane11 and add it to JTabbedPane
    JLabel label1 = new JLabel( "panel one", SwingConstants.CENTER );
    JPanel panel1 = new JPanel(); // create first panel
    panel1.add( label1 ); // add label to panel
    tabbedPane.addTab( "Tab One", null, panel1, "First Panel" );

    // set up panel2 and add it to JTabbedPane
    JLabel label2 = new JLabel( "panel two", SwingConstants.CENTER );
    JPanel panel2 = new JPanel(); // create second panel
    panel2.setBackground( Color.YELLOW ); // set background to yellow
    panel2.add( label2 ); // add label to panel
    tabbedPane.addTab( "Tab Two", null, panel2, "Second Panel" );

    // set up panel3 and add it to JTabbedPane
    JLabel label3 = new JLabel( "panel three" );
    JPanel panel3 = new JPanel(); // create third panel
    panel3.setLayout( new BorderLayout() ); // use borderlayout
    panel3.add( new JButton( "North" ), BorderLayout.NORTH );
```

```
    panel3.add( new JButton( "West" ), BorderLayout.WEST );
    panel3.add( new JButton( "East" ), BorderLayout.EAST );
    panel3.add( new JButton( "South" ), BorderLayout.SOUTH );
    panel3.add( label3, BorderLayout.CENTER );
    tabbedPane.addTab( "Tab Three", null, panel3, "Third Panel" );

    add( tabbedPane ); // add JTabbedPane to frame
  } // end JTabbedPaneFrame constructor
} // end class JTabbedPaneFrame


// JTabbedPaneDemo.java
// Demonstrating JTabbedPane.
import javax.swing.JFrame;

public class JTabbedPaneDemo
{
  public static void main( String[] args )
  {
    JTabbedPaneFrame tabbedPaneFrame = new JTabbedPaneFrame();
    tabbedPaneFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    tabbedPaneFrame.setSize( 250, 200 ); // set frame size
    tabbedPaneFrame.setVisible( true ); // display frame
  } // end main
} // end class JTabbedPaneDemo
```

output



Tables

A table is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable class**, which extends JComponent.

One of its **constructors** is shown here:
JTable(Object data[ ][ ], Object colHeads[ ])
Here, **data is a two-dimensional array of the information to be presented**, and **colHeads is a one-dimensional array with the column headings.**
Here are the steps for using a table in a Frame:
1. Create a JTable object.

2. Create a JScrollPane object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to JFrame

```
//TableFrame.java
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import java.awt.BorderLayout;
import javax.swing.*;

public class TableFrame extends JFrame{
public TableFrame()
   {
super("Testing Table");
setLayout(new BorderLayout());
// Initialize column headings
final String[] colHeads = { "Name", "Phone", "Fax" };
// Initialize data
final Object[][] data = {
{ "Gail", "4567", "8675" },
{ "Ken", "7566", "5555" },
{ "Viviane", "5634", "5887" },
{ "Melanie", "7345", "9222" },
{ "Anne", "1237", "3333" },
{ "John", "5656", "3144" },
{ "Matt", "5672", "2176" },
{ "Claire", "6741", "4244" },
{ "Erwin", "9023", "5159" },
{ "Ellen", "1134", "5332" },
{ "Jennifer", "5689", "1212" },
{ "Ed", "9030", "1313" },
{ "Helen", "6751", "1415" }
};
// Create the table
JTable table = new JTable(data, colHeads);
// Add table to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane scrollPane = new JScrollPane(table, v, h);
// Add scroll pane to Frame
add(scrollPane, BorderLayout.CENTER);
}
}

//TableDemo.java
import javax.swing.JFrame;
```

```
public class TableDemo
{
  public static void main( String[] args )
  {
    TableFrame tableFrame = new TableFrame();
    tableFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    tableFrame.setSize( 250, 200 ); // set frame size
    tableFrame.setVisible( true ); // display frame
  } // end main
} // end class Table
```

output

Trees

A tree is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree class,** which extends JComponent.

A JTree object generates events when a node is expanded or collapsed. The **addTreeExpansionListener( ) and removeTreeExpansionListener( )** methods allow listeners to register and unregister for these notifications. The signatures of these methods are :

void addTreeExpansionListener(TreeExpansionListener tel)
void removeTreeExpansionListener(TreeExpansionListener tel)

The **getPathForLocation( )** method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is shown here:

TreePath getPathForLocation(int x, int y)

Here, x and y are the coordinates at which the mouse is clicked. The return value is a TreePath object that encapsulates information about the tree node that was selected by the user.

The **TreeNode interface** declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The **MutableTreeNode interface extends TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.

The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is shown here:

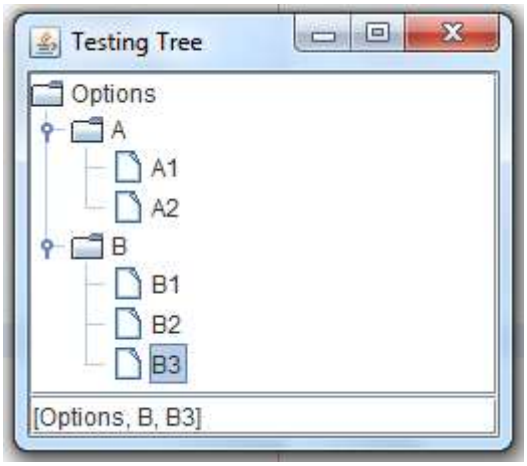
DefaultMutableTreeNode(Object obj)
Here, obj is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.
To create a hierarchy of tree nodes, the **add( )** method of DefaultMutableTreeNode can be used. Its signature is shown here:
void add(MutableTreeNode child)
Here, child is a mutable tree node that is to be added as a child to the current node.
Here are the steps that should be followed to use a tree in a JFrame:
1. Create a JTree object.
2. Create a JScrollPane object. (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the JFrame.

```
//TreeFrame.java
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import java.awt.BorderLayout;
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class TreeFrame extends JFrame{

private JTextField textField   ;
private JTree tree;
public TreeFrame(){
super("Testing Tree");
setLayout(new BorderLayout());
// Create top node of tree
DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
// Create subtree of "A"
DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
top.add(a);
DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
a.add(a1);
DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
a.add(a2);
// Create subtree of "B"
DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
top.add(b);
DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
b.add(b1);
DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
b.add(b2);
DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
b.add(b3);
```

```
// Create tree
 tree = new JTree(top);
// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane scrollPane = new JScrollPane(tree, v, h);
// Add scroll pane to Frame
add(scrollPane, BorderLayout.CENTER);
// Add text field to Frame
 textField  = new JTextField("", 20);
add(textField , BorderLayout.SOUTH);
// Anonymous inner class to handle mouse clicks
tree.addMouseListener(new MouseAdapter() {
public void mouseClicked(MouseEvent me) {
doMouseClicked(me);
}
});
   }
void doMouseClicked(MouseEvent me) {
TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
if(tp != null)
textField.setText(tp.toString());
else
textField.setText("");
}
}
```

```
//TreeDemo.java
import javax.swing.JFrame;

public class TreeDemo
{
  public static void main( String[] args )
  {
    TreeFrame treeFrame = new TreeFrame();
     treeFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
     treeFrame.setSize( 250, 200 ); // set frame size
     treeFrame.setVisible( true ); // display frame
  } // end main
} // end class Tree
```

output



Creating GUI components in NetBeans using drag and drop

Step 1: Create a New Project

Step 2: Choose General -> Java Application

Step 3: Set a Project Name"CelsiusConverterProject". Make sure to deselect the "Create Main Class" checkbox; leaving this option selected generates a new class as the main entry point for the application, but our main GUI window (created in the next step) will serve that purpose, so checking this box is not necessary. Click the "Finish" button when you are done.

Step 4: Add a JFrame Form-Now right-click the CelsiusConverterProject name and choose New -> JFrame Form (JFrame is the Swing class responsible for the main frame for your application.)

Step 5: Name the GUI Class-Next type CelsiusConverterGUI as the class name, and CelsiusConverter as the package name. The remainder of the fields should automatically be filled in, as shown above. Click the Finish button when you are done.

When the IDE finishes loading, the right pane will display a design-time, graphical view of the CelsiusConverterGUI. It is on this screen that you will visually drag, drop, and manipulate the various Swing components.

NetBeans IDE Basics

It is not necessary to learn every feature of the NetBeans IDE before exploring its GUI creation capabilities. In fact, the only features that you really need to understand are **the Palette, the Design Area, the Property Editor, and the Inspector.**

The Palette

The Palette contains all of the components offered by the Swing API(JLabel is a text label, JList is a drop-down list, etc.).

The Design Area

The Design Area is where you will visually construct your GUI. It has two views: **source view, and design view.** Design view is the default, as shown below. You can toggle between views at any time by clicking their respective tabs.

The Property Editor

The Property Editor does what its name implies: it allows you to edit the properties of each component. The Property Editor is intuitive to use; in it you will see a series of rows — one row per property — that you can click and edit without entering the source code directly.

The Inspector

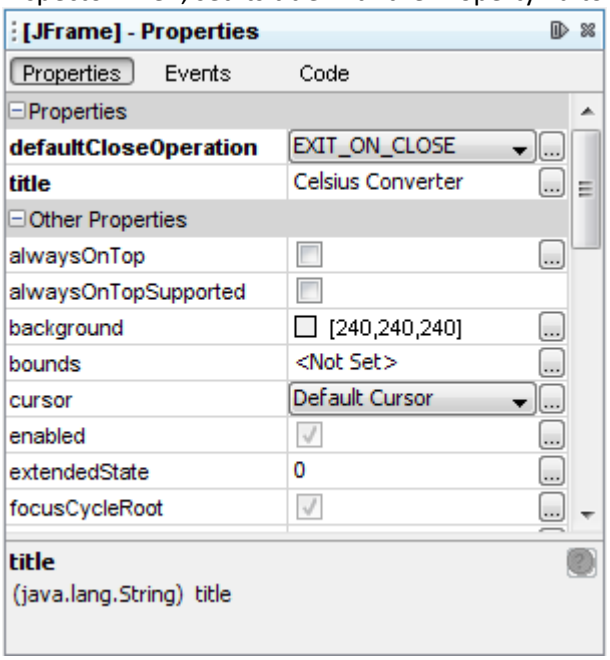
The Inspector provides a graphical representation of your application's components. We will use the Inspector only once, to change a few variable names to something other than their defaults.

Creating the CelsiusConverter GUI

This section explains how to use the NetBeans IDE to create the application's GUI. As you drag each component from the Palette to the Design Area, the IDE auto-generates the appropriate source code.

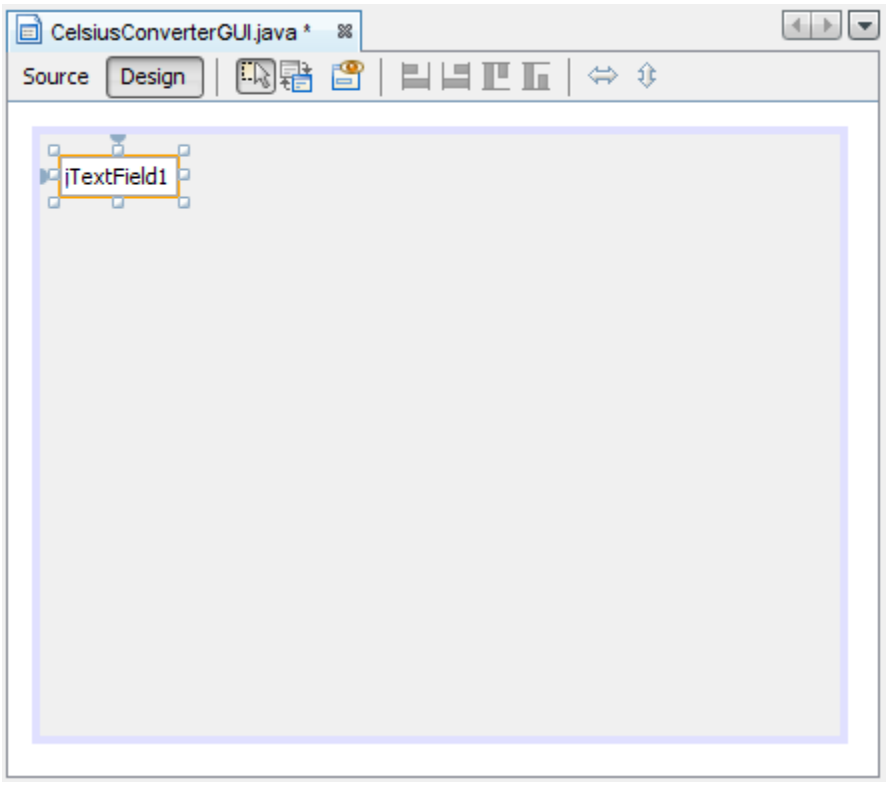
Step 1: **Set the Title**

First, set the title of the application's JFrame to "Celsius Converter", by single-clicking the JFrame in the Inspector:Then, set its title with the Property Editor:

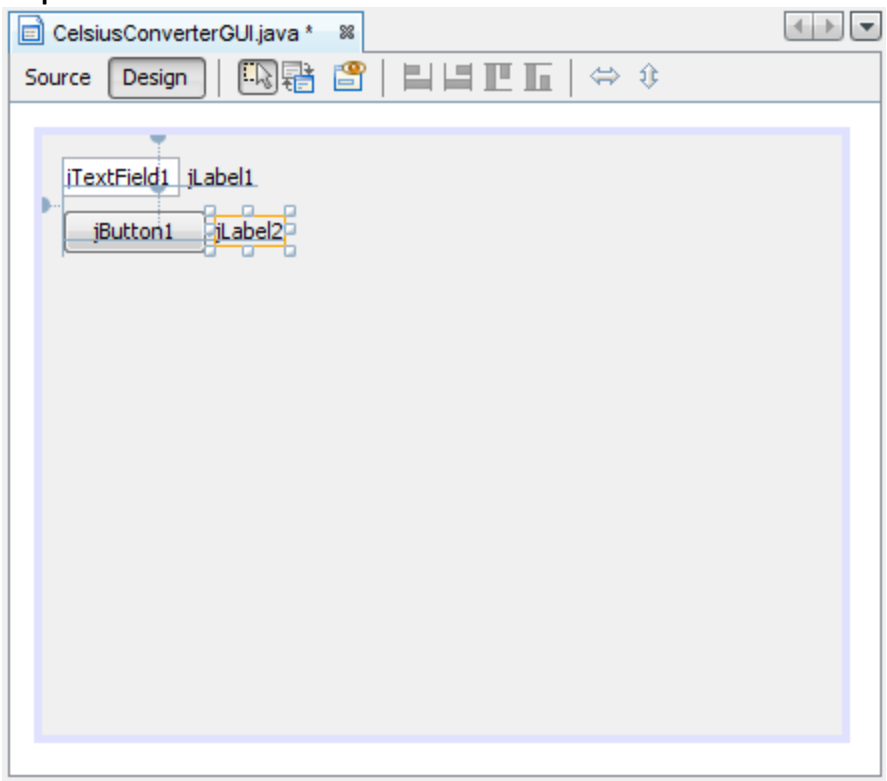


Step 2: Add a JTextField

Next, drag a JTextField from the Palette to the upper left corner of the Design Area. As you approach the upper left corner, the GUI builder provides visual cues (dashed lines) that suggest the appropriate spacing. Using these cues as a guide, drop a JTextField into the upper left hand corner of the window as shown below:



Step 3: Add a JLabel
Step 4: Add a JButton
Step 5: Add a Second JLabel



Adjusting the CelsiusConverter GUI

Step 1: Set the Component Text

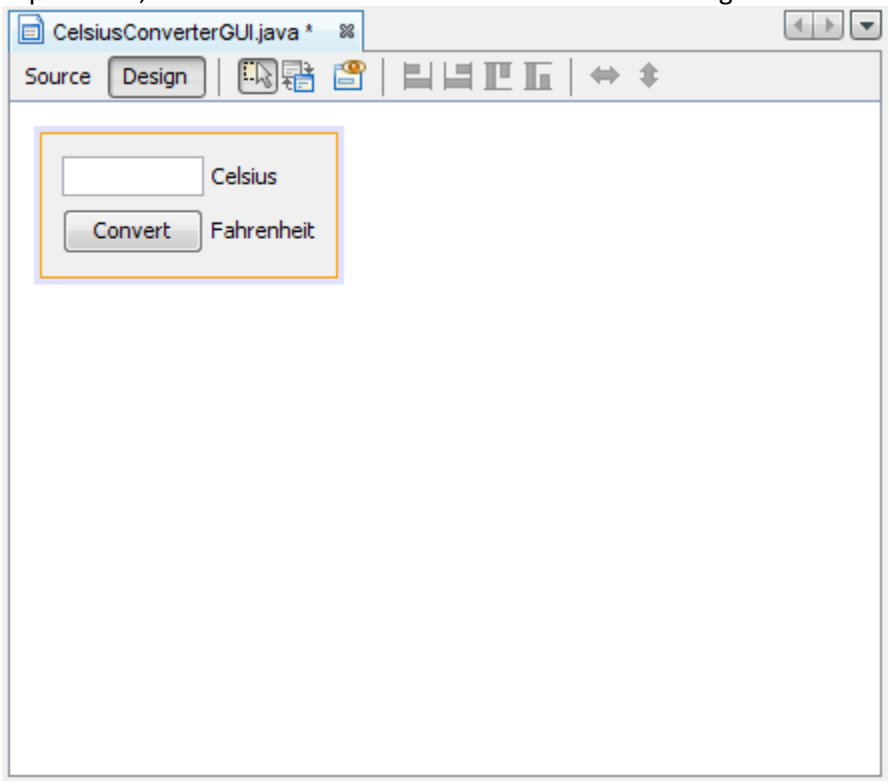
First, double-click the JTextField and JButton to change the default text that was inserted by the IDE. When you erase the text from the JTextField, it will shrink in size as shown below. Change the text of the JButton from "JButton1" to "Convert." Also change the top JLabel text to "Celsius" and the bottom to "Fahrenheit."

Step 2: Set the Component Size

Next, shift-click the JTextField and JButton components. This will highlight each showing that they are selected. Right-click (control-click for mac users) Same Size -> Same Width. The components will now be the same width, as shown below. When you perform this step, make sure that JFrame itself is not also selected. If it is, the Same Size menu will not be active.

Step 3: Remove Extra Space

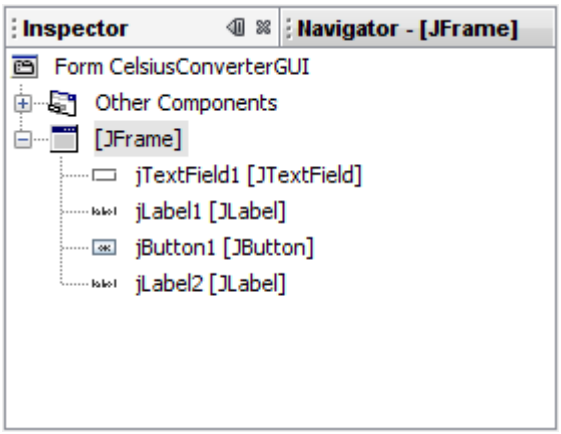
Finally, grab the lower right-hand corner of the JFrame and adjust its size to eliminate any extra whitespace. Note that if you eliminate all of the extra space (as shown below) the title (which only appears at runtime) may not show completely. The end-user is free to resize the application as desired, but you may want to leave some extra space on the right side to make sure that everything fits correctly. Experiment, and use the screenshot of the finished GUI as a guide.



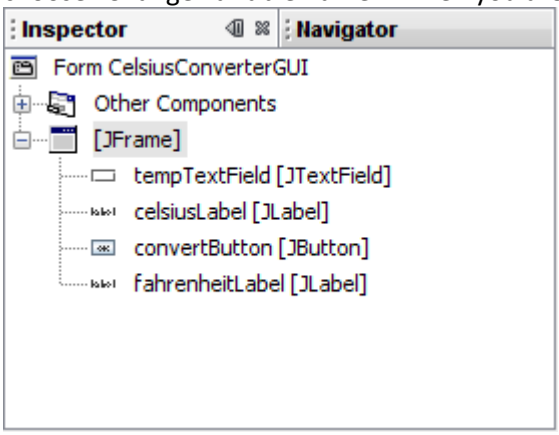
Adding the Application Logic

Step 1: Change the Default Variable Names

The figure below shows the default variable names as they currently appear within the Inspector. For each component, the variable name appears first, followed by the object's type in square brackets. For example, jTextField1 [JTextField] means that "jTextField1" is the variable name and "JTextField" is its type.

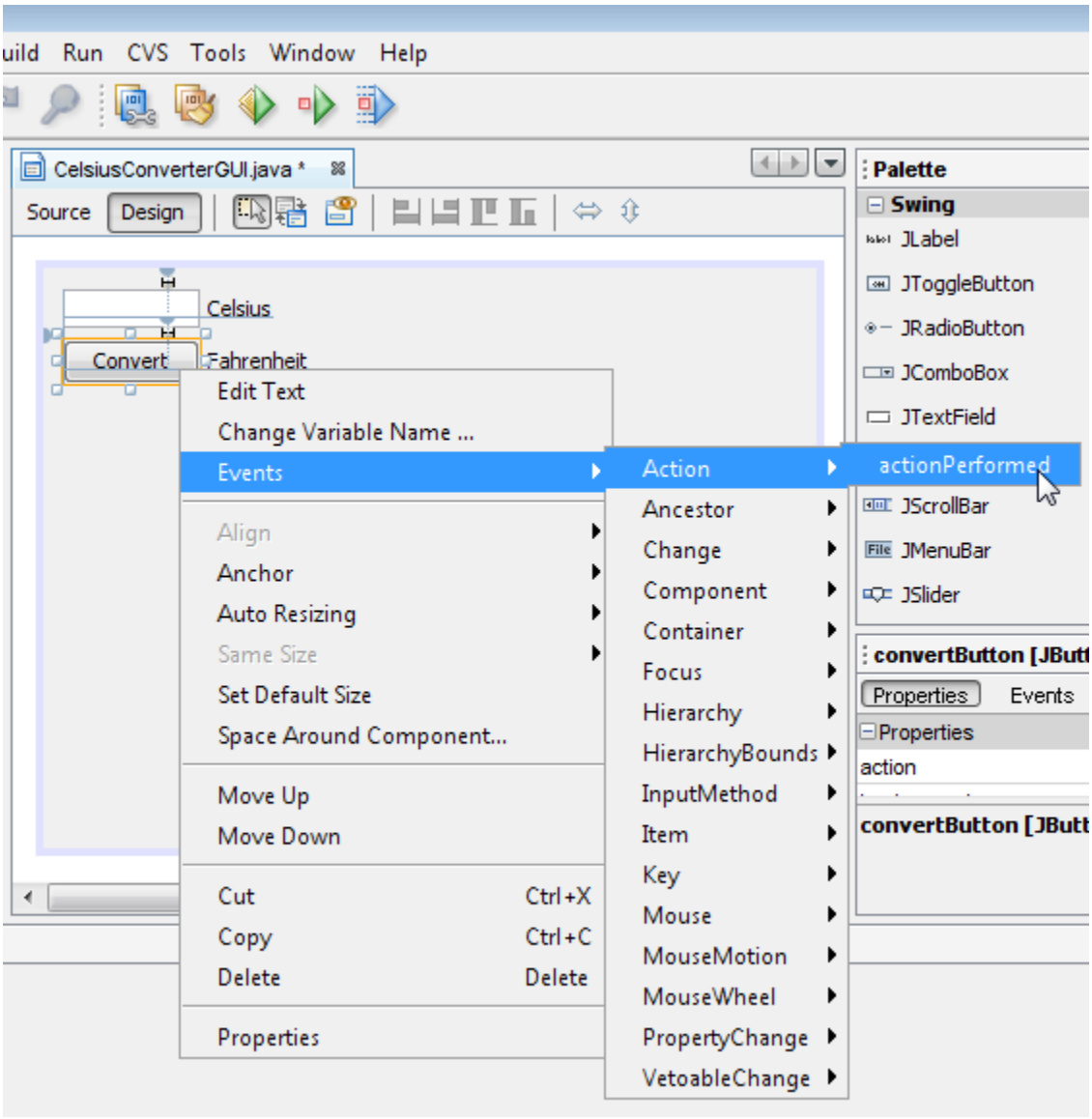


The default names are not very relevant in the context of this application, so it makes sense to change them from their defaults to something that is more meaningful. Right-click each variable name and choose "Change variable name." When you are finished, the variable names should appear as follows:

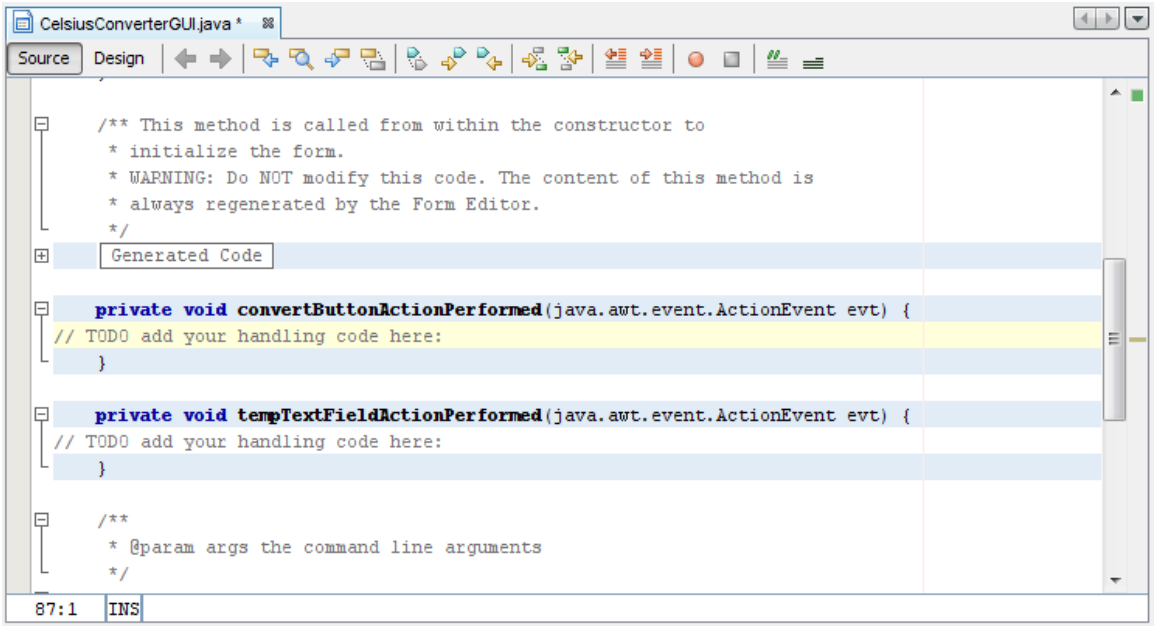


Step 2: Register the Event Listeners

When an end-user interacts with a Swing GUI component (such as clicking the Convert button), that component will generate a special kind of object — called an event object — which it will then broadcast to any other objects that have previously registered themselves as listeners for that event. The NetBeans IDE makes event listener registration extremely simple:



In the Design Area, click on the Convert button to select it. Make sure that only the Convert button is selected (if the JFrame itself is also selected, this step will not work.) Right-click the Convert button and choose Events -> Action -> ActionPerformed. This will generate the required event-handling code, leaving you with empty method bodies in which to add your own functionality:

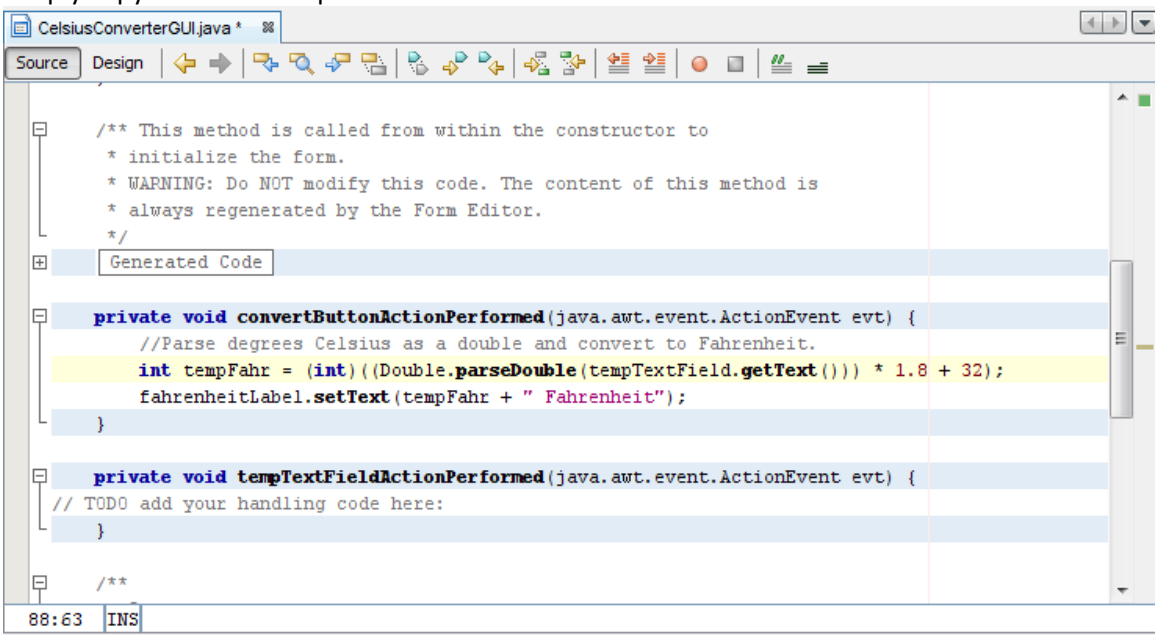


Step 3: Add the Temperature Conversion Code

The final step is to simply paste the temperature conversion code into the empty method body.

```
//Parse degrees Celsius as a double and convert to Fahrenheit.
int tempFahr = (int)((Double.parseDouble(tempTextField.getText()))
        * 1.8 + 32);
fahrenheitLabel.setText(tempFahr + " Fahrenheit");
```

Simply copy this code and paste it into the convertButtonActionPerformed method as shown below:



Step 4: Run the Application

Unit 3- Database Connectivity

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** form any users. **Database management systems allow for the access and storage of data without concern for the internal representation of data.**

Today's most popular database systems are **relational databases**.A language called **SQL**—pronounced "sequel," or as its individual letters—is the international standard language used almost universally with relational databases to perform **queries** (i.e., to request information that satisfies given criteria) and to manipulate 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 you to manipulate that database using the JDBC API.

JDBC Introduction

The **JDBC API** is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps to write Java applications that manage these three programming activities:

- Connect to a data source, like a database
- Send queries and update statements to the database
- Retrieve and process the results received from the database in answer to your query

JDBC includes four components:

The JDBC API — The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.The JDBC API is part of the Java platform, which includes the Java™ Standard Edition (Java™ SE) and the Java™ Enterprise Edition (Java™ EE). The **JDBC 4.0 API** is divided into **two packages**: **java.sql and javax.sql**. Both packages are included in the Java SE and Java EE platforms.

JDBC Driver Manager — The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

JDBC Test Suite — The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

JDBC-ODBC Bridge — The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

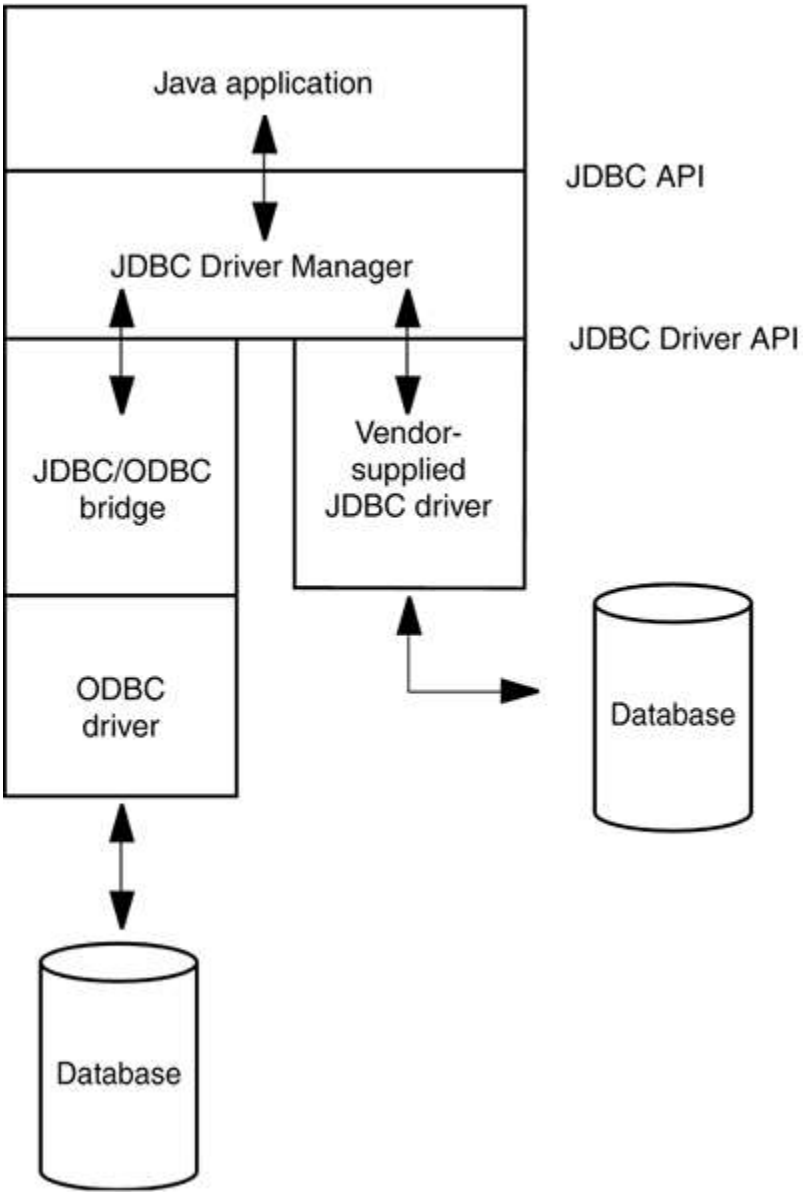


fig.JDBC-to-database communication path

JDBC Driver Types

JDBC drivers are classified into the following types:

- A **type 1 driver** translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Sun includes one such driver, the JDBC/ODBC bridge, with the JDK. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, plenty of better drivers are available, and is advised against using the JDBC/ODBC bridge.

- A **type 2 driver** is written partly in Java and partly in native code; it communicates with the client API of a database. When you use such a driver, you must install some platform-specific code in addition to a Java library.

- A **type 3 driver** is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This can simplify deployment since the database-dependent code is located only on the server.

- A **type 4 driver** is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers that are provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:
Programmers can write applications in the Java programming language to access any database, using standard SQL statementsor even specialized extensions of SQLwhile still following Java language conventions.
Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

Typical Uses of JDBC

The traditional **client/server** model has a rich **GUI on the client** and **a database on the server (**figure below). In this model, a JDBC driver is deployed on the client.

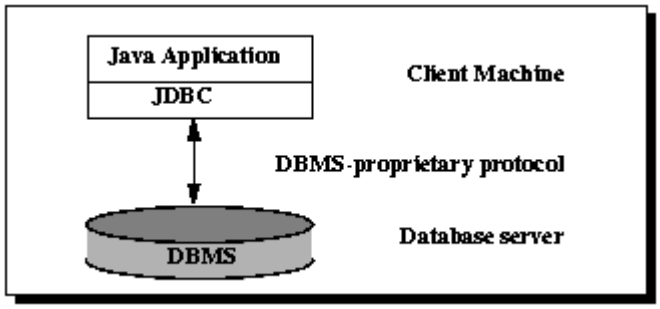


fig.Two-tier Architecture for Data Access.

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 you use a web browser as the client), RMI (when you use an application), or another mechanism. JDBC manages the communication between the middle tier and the back-end database. Figure below shows the basic three tier architecture.

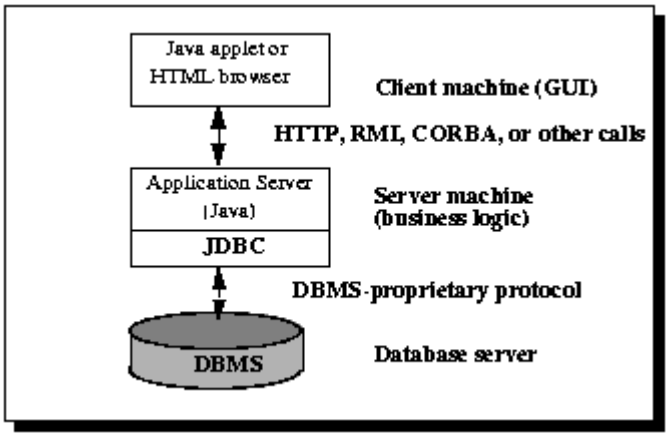


fig.Three-tier Architecture for Data Access

Database connections in Java using JDBC and Netbeans IDE

In order to make a JDBC connection to MySQL database one needs to downlad the **MySQL Connector/J**. It is also expected that you have Netbeans 7+ installed on your machine.

Extract the zip file to a folder, you'll see file '**mysql-connector-java-5.1.20-bin.jar**' which is the library file that we want. Just copy the file to the library folder, for example to **"C:\Program Files\Java\jdk1.7\lib" also to the "C:\Program Files\Java\jdk1.7\jre\lib directory.**

Next, create a **new Java project** on NetBeans named 'DatabaseConnectivity'.
Right click the project and select '**properties**' then under the '**categories'** click on **'libraries'** and click on the '**Add JAR/Folder'** and then browse to **"C:\Program Files\Java\jdk1.7\lib\mysql-connector-java-5.1.20-bin.jar" ,** click on **'open** ' and **'ok'.**

In the services tab of the netbeans , right click the **database** and click on '**new connection'**. Under the **'new connection wizard'**, click on the **'Driver Combobox'** and select the **'mySQL Connector/J driver'** and click on **'next'** untill the wizard completes.

Instructions for Setting Up a MySQL User Account

For the MySQL examples to execute correctly, you need to set up a user account that allows
users to create, delete and modify a database. After MySQL is installed, follow the (these steps assume MySQL is installed in its default installation directory):

1. Open a Command Prompt and start the database server by executing the command **mysqld.exe**. This command has no output—it simply starts the MySQL server. Do not close this window—doing so terminates the server.
2. Next, you'll start theMySQL monitor so you can set up a user account, open another Command Prompt and execute the command
mysql -h localhost -u root
The -h option indicates the host (i.e., computer) on which the MySQL server is running—in this case your local computer (localhost). The -u option indicates the user account that will be used to log in to the server—root is the default user account that is created during installation to allow you to configure the server. Once you've logged in, you'll see a mysql> prompt at which you can type commands to interact with the MySQL server.
3.At the mysql> prompt, type
USE mysql;
and press Enter to select the built-in database named mysql, which stores server information, such as user accounts and their privileges for interacting with the server. Each command must end with a semicolon. To confirm the command, MySQL issues the message "Database changed."
4. Next, you'll add the user account to the mysql built-in database. The mysql database contains a table called user with columns that represent the user's name, password and various privileges. To create the user account '**abc**'with the password **'pqr'** , execute the following commands from the mysql> prompt:
create user 'abc'@'localhost' identified by 'pqr';
grant select, insert, update, delete, create, drop, references,
execute on *.* to 'abc'@'localhost';
5.Type the command
exit;
to terminate the MySQL monitor.

Connecting to and Querying a Database

The example below 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/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

output

```
Authors Table of Books Database:

AuthorID        FirstName       LastName
1               Harvey          Deitel
2               Paul            Deitel
3               Andrew          Goldberg
4               David           Choffnes
```

Connecting to the Database

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 get-Connection 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/books specifies the **protocol for communication (jdbc), the subprotocol for communication (mysql)** and **the location of the database (//localhost/books, where localhost is the host running the MySQL server and books is the database name).** The subprotocol mysql indicates that the program uses a MySQL-specific subprotocol to connect to the MySQL database. If the DriverManager cannot connect to the database, method getConnection throws a **SQLException (package java.sql**).

Figure below 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*

Creating a Statement for Executing Queries

Connection method **createStatement** is invoked to obtain an object that implements **interface Statement** (package java.sql). The program uses the Statement object to submit SQL statements to the database.

Executing a Query

The **Statement** object's **executeQuery** method is used 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.

Processing a Query's ResultSet

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 is used to retrieve the number of columns in the ResultSet.

Retrieving and Modifying Values from Result Sets

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. A ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

You access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method **ResultSet.next** moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next method with a while loop to iterate through all the data in the ResultSet.

ResultSet Interface

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are **type, concurrency, and cursor holdability.**

ResultSet Types

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object.

The sensitivity of a ResultSet object is determined by one of three different ResultSet types:

(a)TYPE_FORWARD_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

(b)TYPE_SCROLL_INSENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

(c)TYPE_SCROLL_SENSITIVE: The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

Note: Not all databases and JDBC drivers support all ResultSet types. The method DatabaseMetaData.supportsResultSetType returns true if the specified ResultSet type is supported and false otherwise.

ResultSet Concurrency

The concurrency of a ResultSet object determines what level of update functionality is supported.

There are two concurrency levels:

CONCUR_READ_ONLY: The ResultSet object cannot be updated using the ResultSet interface.

CONCUR_UPDATABLE: The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR_READ_ONLY.

Note: Not all JDBC drivers and databases support concurrency. The method DatabaseMetaData.supportsResultSetConcurrency returns true if the specified concurrency level is supported by the driver and false otherwise.

Cursor Holdability

Calling the method Connection.commit can close the ResultSet objects that have been created during the current transaction. In some cases, however, this may not be the desired behavior. The ResultSet property holdability gives the application control over whether ResultSet objects (cursors) are closed when commit is called.

The following ResultSet constants may be supplied to the Connection methods createStatement, prepareStatement, and prepareCall:

HOLD_CURSORS_OVER_COMMIT: ResultSet cursors are not closed; they are holdable: they are held open when the method commit is called. Holdable cursors might be ideal if your application uses mostly read-only ResultSet objects.

CLOSE_CURSORS_AT_COMMIT: ResultSet objects (cursors) are closed when the commit method is called. Closing cursors when this method is called can result in better performance for some applications.

The default cursor holdability varies depending on your DBMS.

Retrieving Column Values from Rows

The ResultSet interface declares **getter methods** (for example, **getBoolean and getLong**) for retrieving column values from the current row. You can retrieve values using either the index number of the column or the alias or name of the column. The column index is usually more efficient. Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

```
try
{

  // 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() )
```

```
    {
        int id = rs.getInt("AuthorID");
        String firstName = rs.getString("FirstName");
        String lastName = rs.getString("LastName");
        System.out.println(id+ "\t" + firstName+
                "\t" + lastName );
    } // end while
  } // end try
  catch ( SQLException sqlException )
  {
    sqlException.printStackTrace();
  } // end catch
```

Cursors

As mentioned previously, you access the data in a **ResultSet object through a cursor**, which points to one row in the ResultSet object. However, when a ResultSet object is first created, the cursor is positioned before the first row.There are other methods available to move the cursor:

next: Moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.

previous: Moves the cursor backward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.

first: Moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.

last: Moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.

beforeFirst: Positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.

afterLast: Positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.

relative(int rows): Moves the cursor relative to its current position.

absolute(int row): Positions the cursor on the row specified by the parameter row.

Note that the default sensitivity of a ResultSet is TYPE_FORWARD_ONLY, which means that it cannot be scrolled; you cannot call any of these methods that move the cursor, except next, if your ResultSet cannot be scrolled.

Updating Rows in ResultSet Objects

You cannot update a default ResultSet object, and you can only move its cursor forward. However, you can create ResultSet objects that can be scrolled (the cursor can move backwards or move to an absolute position) and updated.

```
try {
    // establish connection to database
     connection = DriverManager.getConnection(
      DATABASE_URL, "root", "" );
    statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
          ResultSet.CONCUR_UPDATABLE);
    ResultSet uprs= statement.executeQuery(
```

```
        "SELECT * FROM authors");

      while (uprs.next()) {
        uprs.updateString( "LastName","Sharma");
        uprs.updateRow();
      }
      }
    catch ( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch
```

The field **ResultSet.TYPE_SCROLL_SENSITIVE** creates a ResultSet object whose cursor can move both forward and backward relative to the current position and to an absolute position. The field **ResultSet.CONCUR_UPDATABLE** creates a ResultSet object that can be updated. See the ResultSet Javadoc for other fields you can specify to modify the behavior of ResultSet objects.

The method **ResultSet.updateString** updates the specified column (in this example, LastName with the specified float value in the row where the cursor is positioned. ResultSet contains various updater methods that enable you to update column values of various data types. However, none of these updater methods modifies the database; you must call the method **ResultSet.updateRow to update the database**.

Inserting Rows in ResultSet Objects

Note: Not all JDBC drivers support inserting new rows with the ResultSet interface. If you attempt to insert a new row and your JDBC driver database does not support this feature, a SQLFeatureNotSupportedException exception is thrown.

```
try {
      statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);

      ResultSet uprs = statement.executeQuery(
        "SELECT * FROM authors");

      uprs.moveToInsertRow();
      uprs.updateInt("AuthorID",9);
      uprs.updateString("FirstName","Subash");
      uprs.updateString("LastName","Pakhrin");
      uprs.insertRow();
      uprs.beforeFirst();
   }
      catch ( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch
```

This example calls the Connection.createStatement method with two arguments, **ResultSet.TYPE_SCROLL_SENSITIVE and ResultSet.CONCUR_UPDATABLE**. The first value enables the

cursor of the ResultSet object to be moved both forward and backward. The second value, ResultSet.CONCUR_UPDATABLE, is required if you want to insert rows into a ResultSet object; it specifies that it can be updatable.

The same stipulations for using strings in getter methods also apply to updater methods.

The method **ResultSet.moveToInsertRow** moves the cursor to the insert row. The insert row is a special row associated with an updatable result set. It is essentially **a buffer** where a new row can be constructed by calling the updater methods prior to inserting the row into the result set. For example, this method calls the method ResultSet.updateString to update the insert row's COF_NAME column to Kona.

The method ResultSet.insertRow inserts the contents of the insert row into the ResultSet object and into the database.

Note: After inserting a row with the ResultSet.insertRow, you should move the cursor to a row other than the insert row. For example, this example moves it to before the first row in the result set with the method **ResultSet.beforeFirst**. Unexpected results can occur if another part of your application uses the same result set and the cursor is still pointing to the insert row.

Using Statement Objects for Batch Updates

Statement, PreparedStatement and CallableStatement objects have a list of commands that is associated with them. This list may contain statements for updating, inserting, or deleting a row; and it may also contain DDL statements such as CREATE TABLE and DROP TABLE. It cannot, however, contain a statement that would produce a ResultSet object, such as a SELECT statement. In other words, the list can contain only statements that produce an update count.

The list, which is associated with a Statement object at its creation, is initially empty. You can add SQL commands to this list with the method addBatch and empty it with the method clearBatch. When you have finished adding statements to the list, call the method **executeBatch** to send them all to the database to be executed as a unit, or batch.

```
try {
   connection = DriverManager.getConnection(
     DATABASE_URL, "root", "" );
 connection.setAutoCommit(false);
 statement = connection.createStatement();

 statement.addBatch(
    "INSERT INTO authors " +
    "VALUES('15','Hari','Shrestha')");

 statement.addBatch(
    "INSERT INTO authors " +
    "VALUES('16','Ram','Acharya')");

 statement.addBatch(
    "INSERT INTO authors  " +
```

```
        "VALUES('17','Shyam','Gautam')");

    statement.addBatch(
        "INSERT INTO authors  " +
        "VALUES('18','Govinda','Paudel')");

     int [] updateCounts = statement.executeBatch();
    connection.commit();

  } catch(BatchUpdateException b) {
     b.printStackTrace();
  } catch(SQLException ex) {
     ex.printStackTrace();
  }
```

The following line disables auto-commit mode for the Connection object con so that the **transaction** will not be automatically committed or rolled back when the method executeBatch is called.
connection.setAutoCommit(false);
To allow for correct error handling, you should always disable auto-commit mode before beginning a batch update.
The method **Statement.addBatch** adds a command to the list of commands associated with the Statement object statement. In this example, these commands are all INSERT INTO statements, each one adding a row consisting of three column values.

The following line sends the four SQL commands that were added to its list of commands to the database to be executed as a batch:

int [] updateCounts = statement.executeBatch();

Note that **statement** uses the method **executeBatch** to send the batch of insertions, **not the method executeUpdate**, **which sends only one command and returns a single update count.** The DBMS executes the commands in the order in which they were added to the list of commands, so it will first add the row of values for "Hari" , then add the row for "Ram", then "Shyam" , and finally "Govinda". If all four commands execute successfully, the DBMS will return an update count for each command in the order in which it was executed. The update counts that indicate how many rows were affected by each command are stored in the array updateCounts.

If all four of the commands in the batch are executed successfully, updateCounts will contain four values, all of which are 1 because an insertion affects one row. The list of commands associated with stmt will now be empty because the four commands added previously were sent to the database when stmt called the method executeBatch. You can at any time explicitly empty this list of commands with the method clearBatch.

The Connection.commit method makes the batch of updates to the "authors" table permanent. This method needs to be called explicitly because the auto-commit mode for this connection was disabled previously.

The following line enables auto-commit mode for the current Connection object.

connection.setAutoCommit(true);

Now each statement in the example will automatically be committed after it is executed, and it no longer needs to invoke the method commit.

PreparedStatements

A **PreparedStatement** enables you to create **compiled SQL statements** that execute more efficiently than Statements. PreparedStatements can also specify parameters, making them more flexible than Statements—you can execute the same query repeatedly with different
parameter values.

The **PreparedStatement** is derived from the more general class, **Statement**. If you want to execute a Statement object many times, it usually reduces execution time to use a PreparedStatement object instead.

Performing Parameterized Batch Update using PreparedStatement

It is also possible to have a parameterized batch update, as shown in the following code fragment, where con is a Connection object:

```
try {
        connection = DriverManager.getConnection(
          DATABASE_URL, "root", "" );
     connection.setAutoCommit(false);
PreparedStatement pstmt = connection.prepareStatement(
                    "INSERT INTO authors VALUES(?, ?, ?)");
pstmt.setInt(1,19);
pstmt.setString(2, "Navin");
pstmt.setString(3,"Sharma");
pstmt.addBatch();

pstmt.setInt(1,20);
pstmt.setString(2, "Rajesh");
pstmt.setString(3,"Paudel");
pstmt.addBatch();

// ... and so on for each new
// type of coffee

int [] updateCounts = pstmt.executeBatch();
connection.commit();
     }
    catch ( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch
```

The three question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a PreparedStatement, the program must specify the parameter values by using the **Prepared- Statement interface's set methods.**

For the preceding query, parameters are int and strings that can be set with Prepared- Statement method **setInt and setString.**
Method setInt's and setString's **first argument represents the parameter number being set, and the second argument is that parameter's value**. Parameter numbers are counted from 1, starting with the first question mark (?).
Interface PreparedStatement provides set methods for each supported SQL type. **It's important to use the set method that is appropriate for the parameter's SQL type in the database—SQLExceptions occur when a program attempts to convert a parameter value to an incorrect type.**

Transaction Processing
Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine if the transaction is successful. You begin by specifying the source account and the amount you wish to transfer from that account to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account.The way to be sure that either both actions occur or neither action occurs is to use a **transaction. A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.**

The way to allow two or more statements to be **grouped into a transaction** is to **disable the auto-commit mode.**
Disabling Auto-Commit Mode
When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.
The way to allow two or more statements to be **grouped into a transaction** is to **disable the auto-commit mode.**
con.setAutoCommit(false);
Committing Transactions
After the auto-commit mode is disabled, no SQL statements are committed until you call the **method commit explicitly**. All statements executed after the previous call to the method commit are included in the **current transaction and committed together as a unit**.
con.commit();
Rollback
If you group update statements to a transaction, then the transaction either succeeds in its entirety and it can be committed, or it fails somewhere in the middle. In that case, you can carry out a **rollback** and **the database automatically undoes the effect of all updates that occurred since the last committed transaction**.

You turn off autocommit mode with the command
conn.setAutoCommit(false);
Now you create a statement object in the normal way:
Statement stat = conn.createStatement();
Call executeUpdate any number of times:
stat.executeUpdate(command1);

stat.executeUpdate(command2);
stat.executeUpdate(command3);
. . .
When all commands have been executed, call the commit method:
conn.commit();
However, if an error occurred, call
conn.rollback();
Then, all commands until the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a SQLException.

Save Points
You can gain finer-grained control over the rollback process by using save points. Creating a save point marks a point to which you can later return without having to return to the start of the transaction. For example,

```
Statement stat = conn.createStatement();     // start transaction; rollback() goes here
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint();        // set savepoint; rollback(svpt) goes here
stat.executeUpdate(command2);
if (. . .) conn.rollback(svpt);              // undo effect of command2
. . .
conn.commit();
```

Here, we used an anonymous save point. You can also give the save point a name, such as
Savepoint svpt = conn.setSavepoint("stage1");
When you are done with a save point, **you should release it:**
stat.releaseSavepoint(svpt);

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.BatchUpdateException;
import java.sql.PreparedStatement;


public class DisplayAuthors
{
  // database URL
  static final String DATABASE_URL = "jdbc:mysql://localhost/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

  try {
   // establish connection to database
     connection = DriverManager.getConnection(
       DATABASE_URL, "root", "" );
   statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
   ResultSet uprs= statement.executeQuery(
       "SELECT * FROM authors");

   while (uprs.next()) {
            uprs.updateString( "LastName","Sharma");
      uprs.updateRow();
   }
   }
 catch ( SQLException sqlException )
  {
    sqlException.printStackTrace();
  } // end catch

  try {
   statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);

   ResultSet uprs = statement.executeQuery(
      "SELECT * FROM authors");

   uprs.moveToInsertRow();
   uprs.updateInt("AuthorID",9);
   uprs.updateString("FirstName","Subash");
   uprs.updateString("LastName","Pakhrin");
   uprs.insertRow();
   uprs.beforeFirst();
}
   catch ( SQLException sqlException )
  {
    sqlException.printStackTrace();
  } // end catch
  try {
           connection.setAutoCommit(false);
   statement = connection.createStatement();

  statement.addBatch(
      "INSERT INTO authors " +
      "VALUES('15','Hari','Shrestha')");

   statement.addBatch(
```

```
            "INSERT INTO authors " +
            "VALUES('16','Ram','Acharya')");

          statement.addBatch(
            "INSERT INTO authors  " +
            "VALUES('17','Shyam','Gautam')");

         statement.addBatch(
            "INSERT INTO authors  " +
            "VALUES('18','Govinda','Paudel')");

          int [] updateCounts = statement.executeBatch();
         connection.commit();

       } catch(BatchUpdateException b) {
          b.printStackTrace();
       } catch(SQLException ex) {
          ex.printStackTrace();
       }
        try {
           connection.setAutoCommit(false);
PreparedStatement pstmt = connection.prepareStatement(
                        "INSERT INTO authors VALUES(?, ?, ?)");
pstmt.setInt(1,19);
pstmt.setString(2, "Navin");
pstmt.setString(3,"Sharma");
pstmt.addBatch();

pstmt.setInt(1,20);
pstmt.setString(2, "Rajesh");
pstmt.setString(3,"Paudel");
pstmt.addBatch();

// ... and so on for each new
// type of authors

int [] updateCounts = pstmt.executeBatch();
connection.commit();
        }
       catch ( SQLException sqlException )
        {
          sqlException.printStackTrace();
        } // end catch

        // connect to database books and query database
        try
        {
```

```
      // 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.setAutoCommit(true);
        connection.close();

      } // end try
      catch ( Exception exception )
      {
        exception.printStackTrace();
      } // end catch
    } // end finally
  } // end main
} // end class DisplayAuthors
```

RowSet Interface

A JDBC RowSet object holds tabular data in a way that makes it more flexible and easier to use than a result set.The RowSet interface configures the database connection and prepares query statements automatically.It provides several **set methods** that allow you to specify the properties needed to establish a connection (such as the database URL, user name and password of the database) and create a Statement (such as a query). RowSet also provides several **get methods** that return these properties.

Connected and Disconnected RowSets

There are **two types of RowSet objects—connected and disconnected**. A **connected RowSet** object connects to the database once and remains connected while the object is in use. A **disconnected RowSet** object connects to the database, executes a query to retrieve the data from the database and then closes the connection. A program may change the data in a disconnected RowSet while it's disconnected. Modified data can be updated in the database
after a disconnected RowSet reestablishes the connection with the database.

Package javax.sql.rowset contains two subinterfaces of RowSet—**JdbcRowSet** and **CachedRowSet**. JdbcRowSet, a connected RowSet, acts as a wrapper around a ResultSet object and allows you to scroll through and update the rows in the ResultSet. By default, a ResultSet object is nonscrollable and read only—you must explicitly set the result set type constant to TYPE_SCROLL_INSENSITIVE and set the result set concurrency constant to CONCUR_UPDATABLE to make a ResultSet object scrollable and updatable.
A JdbcRowSet object is scrollable and updatable by default. CachedRowSet, a disconnected
RowSet, caches the data of a ResultSet in memory and disconnects from the database.Like JdbcRowSet, a CachedRowSet object is scrollable and updatable by default. A Cached-RowSet object is also serializable, so it can be passed between Java applications through a network, such as the Internet. However, CachedRowSet has a limitation—the amount of data that can be stored in memory is limited. Package javax.sql.rowset contains three other subinterfaces of RowSet:**WebRowSet,JoinRowSet and FilteredRowSet.**
(For Reference:http://docs.oracle.com/javase/tutorial/jdbc/basics/rowset.html).

JdbcRowSet

Navigating JdbcRowSet Objects

```
JdbcRowSet jdbcRs = new JdbcRowSetImpl();
jdbcRs.absolute(4);
jdbcRs.previous();
```

Updating Column Values

```
jdbcRs.absolute(3);
jdbcRs.updateString("lastName", "Sharma");
jdbcRs.updateRow();
```

Inserting Rows

```
jdbcRs.moveToInsertRow();
jdbcRs.updateInt("Author_ID", 10);
jdbcRs.updateString("FirstName", "Navin");
jdbcRs.updateString("LastName", "Sharma");
jdbcRs.insertRow();
```

Deleting Rows

```
jdbcRs.last();
```

```
jdbcRs.deleteRow();
```

// Program demonstrating JdbcRowSet

```
//JdbcRowSetTest.java
//Displaying the contents of the Authors table using JdbcRowSet.
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import javax.sql.rowset.JdbcRowSet;
import com.sun.rowset.JdbcRowSetImpl; // Sun's JdbcRowSet implementation
public class JdbcRowSetTest
{
// JDBC driver name and database URL
static final String DATABASE_URL = "jdbc:mysql://localhost/books";
static final String USERNAME = "root";
static final String PASSWORD = "";

// constructor connects to database, queries database, processes
 // results and displays results in window
public JdbcRowSetTest()
{
 // connect to database books and query database
try
 {
// specify properties of JdbcRowSet
JdbcRowSet rowSet = new JdbcRowSetImpl();
rowSet.setUrl( DATABASE_URL ); // set database URL
rowSet.setUsername( USERNAME ); // set username
rowSet.setPassword( PASSWORD ); // set password
rowSet.setCommand( "SELECT * FROM Authors" ); // set query
rowSet.execute(); // execute query
// process query results
ResultSetMetaData metaData = rowSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();
System.out.println( "Authors Table of Books Database:\n" );

// display rowset header
for ( int i = 1; i <= numberOfColumns; i++ )
System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
System.out.println();

// display each row
while(rowSet.next())
{
for ( int i = 1; i <= numberOfColumns; i++ )
System.out.printf( "%-8s\t",rowSet.getObject( i ));
} // end while
// close the underlying ResultSet, Statement and Connection
```

```
rowSet.close();
} // end try
catch ( SQLException sqlException )
{
sqlException.printStackTrace();
 System.exit( 1 );
} // end catch
} // end DisplayAuthors constructor
// launch the application
public static void main( String args[] )
{
JdbcRowSetTest application = new JdbcRowSetTest();
} // end main
} // end class JdbcRowSetTest
```

CachedRowSet

Creating CachedRowSet Objects:

```
achedRowSet crs = new CachedRowSetImpl();
```

Setting CachedRowSet Properties:

```
crs.setUsername(username);
crs.setPassword(password);
crs.setUrl("jdbc:mySubprotocol:mySubname");
```

Setting up command:

```
crs.setCommand("select * from Authors");
```

Populating CachedRowSet Objects:

```
crs.execute();
```

Updating CachedRowSet Object:

```
 crs.updateInt("Author_ID",       10);
 crs.updateString("FirstName", "Navin");
            ....................
  crs.updateRow();
// Synchronizing the row back to the DB
    crs.acceptChanges(con);
```

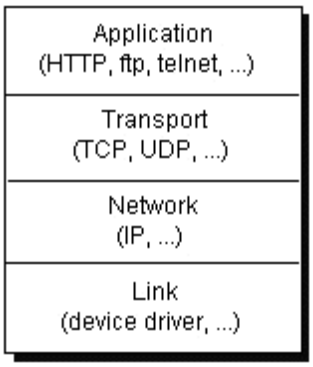
Inserting and Deleting Rows:

```
  crs.absolute(3);
  crs.updateString("lastName", "Sharma");
  crs.updateRow();
  crs.insertRow();
  crs.moveToCurrentRow();
  crs.acceptChanges(con);
```

Unit-5 Network Programming

Networking Basics

Computers running on the Internet communicate to each other using either the **Transmission Control Protocol (TCP)** or the **User Datagram Protocol (UDP),** as this diagram illustrates:



When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the **java.net package**. These classes provide **system-independent network communication**. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

Transmission Control Protocol (TCP)

TCP (Transmission Control Protocol) is a connection-based protocol that provides a reliable flow of data between two computers.

When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection. This is analogous to making a telephone call. If you want to speak to your friend, a connection is established when you dial his phone number and he answers. You send data back and forth over the connection by speaking to one another over the phone lines. Like the phone company, **TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.**

TCP provides a point-to-point channel for applications that require reliable communications. The **Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet** are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, user end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

User Datagram Protocol (UDP)

UDP (User Datagram Protocol) is a protocol that sends independent packets of data, called **datagrams**, from one computer to another with no guarantees about arrival.

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called datagrams, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is

independent of any other.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, **they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.**

Consider, for example, a clock server that sends the current time to its client when requested to do so. If the client misses a packet, it doesn't really make sense to resend it because the time will be incorrect when the client receives it on the second try. If the client makes two requests and receives packets from the server out of order, it doesn't really matter because the client can figure out that the packets are out of order and make another request. The reliability of TCP is unnecessary in this instance because it causes performance degradation and may hinder the usefulness of the service.

Another example of a service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network. In fact, ping needs to know about dropped or out-of-order packets to determine how good or bad the connection is. A reliable channel would invalidate this service altogether.

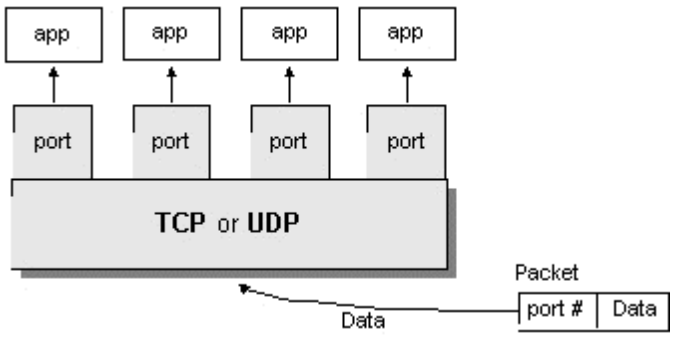
Many firewalls and routers have been configured not to allow UDP packets. If you're having trouble connecting to a service outside your firewall, or if clients are having trouble connecting to your service, you should check whether UDP is permitted.

Ports

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.Generally speaking, a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different applications running on the computer. So how does the computer know to which application to forward the data? Through the use of ports.

Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its **32-bit IP** address, which IP uses to deliver data to the right computer on the network. **Ports** are identified by a **16-bit number**, which TCP and UDP use to deliver the data to the right application.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port.
In datagram-based communication such as UDP, the datagram packet contains the port number of its destination and UDP routes the packet to the appropriate application.



Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers. The port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called well-known ports. Your applications should not attempt to bind to them.

Networking Classes in the JDK

Through the classes in **java.net**, Java programs can use TCP or UDP to communicate over the Internet. The **URL, URLConnection, Socket, and ServerSocket classes** all **use TCP** to communicate over the network. The **DatagramPacket, DatagramSocket, and MulticastSocket classes** are for use with **UDP**.

Working with URLs

URL is the acronym for **Uniform Resource Locator**. It is a reference (an address) to a resource on the Internet. You provide URLs to your favorite Web browser so that it can locate files on the Internet in the same way that you provide addresses on letters so that the post office can locate your correspondents.

Java programs that interact with the Internet also may use URLs to find the resources on the Internet they wish to access. **Java programs can use a class called URL in the java.net package to represent a URL address.**

The term URL can be ambiguous. It can refer to an Internet address or a URL object in a Java program. Here "URL address" is used to mean an Internet address and "URL object" to refer to an instance of the URL class in a program.

URL

URL is an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet.If you've been surfing the Web, you have undoubtedly heard the term URL and have used URLs to access HTML pages from the Web.

It's often easiest, although not entirely accurate, to think of a URL as the name of a file on the World Wide Web because most URLs refer to a file on some machine on the network. However, remember that URLs also can point to other resources on the network, such as database queries and command output.

A URL has **two main components**:

Protocol identifier: For the URL http://example.com, the **protocol identifier is http.**

Resource name: For the URL http://example.com, the **resource name is example.com.**

Note that the protocol identifier and the resource name are separated by a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, **the resource name contains one or more of the following components:**

Host Name

The name of the machine on which the resource lives.

Filename

The pathname to the file on the machine.

Port Number

The port number to which to connect (typically optional).

Reference

A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

Creating a URL

The easiest way to create a URL object is from a String that represents the human-readable form of the URL address. This is typically the form that another person will use for a URL. In your Java program, you can use a String containing this text to create a URL object:

```
URL myURL = new URL("http://example.com/");
```

The URL object created above represents **an absolute URL**. An absolute URL contains all of the information necessary to reach the resource in question. You can also create URL objects from a relative URL address.

Creating a URL Relative to Another

In your Java programs, you can create a URL object from a relative URL specification. For example, suppose you know two URLs at the site example.com:
http://example.com/pages/page1.html
http://example.com/pages/page2.html
You can create URL objects for these pages relative to their common base URL:
http://example.com/pages/ like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:

URL(URL baseURL, String relativeURL)

The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base. If baseURL is null, then this constructor treats relativeURL like an absolute URL specification. Conversely, if relativeURL is an absolute URL specification, then the constructor ignores baseURL.

Other URL Constructors

new URL("http", "example.com", "/pages/page1.html");
This is equivalent to
new URL("http://example.com/pages/page1.html");

The first argument is the protocol, the second is the host name, and the last is the pathname of the file. Note that the filename contains a forward slash at the beginning. This indicates that the filename is specified from the root of the host.

The final URL constructor adds the port number to the list of arguments used in the previous constructor:
URL url = new URL("http", "example.com", 80, "pages/page1.html");
This creates a URL object for the following URL:
http://example.com:80/pages/page1.html
If you construct a URL object using one of these constructors, you can get a String containing the complete URL address by using the URL object's toString method or the equivalent toExternalForm method.

URL addresses with Special characters

Some URL addresses contain special characters, for example the space character. Like this:
http://example.com/hello world/
To make these characters legal they need to be encoded before passing them to the URL constructor.
URL url = new URL("http://example.com/hello%20world");
Encoding the special character(s) in this example is easy as there is only one character that needs encoding, but for URL addresses that have several of these characters or if you are unsure when writing your code what URL addresses you will need to access, you can use the multi-argument constructors of the java.net.URI class to automatically take care of the encoding for you.
URI uri = new URI("http", "example.com", "/hello world/", "");
And then convert the URI to a URL.
URL url = uri.toURL();

MalformedURLException

Each of the four URL constructors throws a MalformedURLException if the arguments to the constructor refer to a null or unknown protocol. Typically, you want to catch and handle this exception by embedding your URL constructor statements in a try/catch pair, like this:

```
try {
   URL myURL = new URL(...);
}
```

```
catch (MalformedURLException e) {
   // exception handler code here
   // ...
}
```

Parsing a URL

The URL class provides several methods that let you query URL objects. You can get the **protocol, authority, host name, port number, path, query, filename, and reference** from a URL using these accessor methods:

getProtocol
Returns the protocol identifier component of the URL.

getAuthority
Returns the authority component of the URL.

getHost
Returns the host name component of the URL.

getPort
Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.

getPath
Returns the path component of this URL.

getQuery
Returns the query component of this URL.

getFile
Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.

getRef
Returns the reference component of the URL.

Note:
Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs. The URL class is somewhat HTTP-centric.

You can use these getXXX methods to get information about the URL regardless of the constructor that you used to create the URL object.

The URL class, along with these accessor methods, frees you from ever having to parse URLs again! Given any string specification of a URL, just create a new URL object and call any of the accessor methods for the information you need. This small example program creates a URL from a string specification and then uses the URL object's accessor methods to parse the URL:

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

public class ParseURL {
   public static void main(String[] args) throws Exception {
```

```
        URL aURL = new URL("http://example.com:80/docs/books/tutorial"
                    + "/index.html?name=networking#DOWNLOADING");

        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("authority = " + aURL.getAuthority());
        System.out.println("host = " + aURL.getHost());
        System.out.println("port = " + aURL.getPort());
        System.out.println("path = " + aURL.getPath());
        System.out.println("query = " + aURL.getQuery());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("ref = " + aURL.getRef());
    }
}
```

Here is the output displayed by the program:

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/books/tutorial/index.html
query = name=networking
filename = /docs/books/tutorial/index.html?name=networking
ref = DOWNLOADING
```

Reading Directly from a URL

After you've successfully created a URL, you can call the URL's **openStream() method** to get a stream from which you can read the contents of the URL. **The openStream() method returns a java.io.InputStream object**, so reading from a URL is as easy as reading from an input stream.

The following small Java program uses openStream() to get an input stream on the URL http://www.google.com.np/. It then opens a **BufferedReader** on the input stream and reads from the BufferedReader thereby reading from the URL. Everything read is copied to the standard output stream:

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

public class URLReader {
    public static void main(String[] args) throws Exception {

        URL oracle = new URL("http://www.google.com.np/");
        BufferedReader in = new BufferedReader(
        new InputStreamReader(oracle.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
```

```
    }
}
```

When you run the program, you should see, scrolling by in your command window, the HTML commands and textual content from the HTML file located at http://www.google.com.np/.

Connecting to a URL

After you've successfully created a URL object, you can call the URL object's **openConnection** method to get a URLConnection object, or one of its protocol specific subclasses, e.g. **java.net.HttpURLConnection**

You can use this URLConnection object to setup parameters and general request properties that you may need before connecting. **Connection to the remote object represented by the URL is only initiated when the URLConnection.connect method is called**. When you do this you are initializing a communication link between your Java program and the URL over the network. For example, the following code opens a connection to the site example.com:

```
try {
    URL myURL = new URL("http://example.com/");
    URLConnection myURLConnection = myURL.openConnection();
    myURLConnection.connect();
}
catch (MalformedURLException e) {
    // new URL() failed
    // ...
}
catch (IOException e) {
    // openConnection() failed
    // ...
}
```

A new URLConnection object is created every time by calling the openConnection method of the protocol handler for this URL.

You are not always required to explicitly call the connect method to initiate the connection. Operations that depend on being connected, like getInputStream, getOutputStream, etc, will implicitly perform the connection, if necessary.

Now that you've successfully connected to your URL, you can use the URLConnection object to perform actions such as reading from or writing to the connection. The next example shows how.

Reading from a URLConnection

The following program performs the same function as the URLReader program shown in Reading Directly from a URL.
However, rather than getting an input stream directly from the URL, this program explicitly retrieves a URLConnection object and gets an input stream from the connection. The connection is opened implicitly by calling getInputStream. Then, like URLReader, this program creates a BufferedReader on the input stream and reads from it.

```
import java.net.*;
```

```
import java.io.*;

public class URLConnectionReader {
   public static void main(String[] args) throws Exception {
      URL oracle = new URL("http://www.oracle.com/");
      URLConnection yc = oracle.openConnection();
      BufferedReader in = new BufferedReader(new InputStreamReader(
                              yc.getInputStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null)
         System.out.println(inputLine);
      in.close();
   }
}
```

The output from this program is identical to the output from the program that opens a stream directly from the URL. You can use either way to read from a URL. However, reading from a URLConnection instead of reading directly from a URL might be more useful. This is because you can use the URLConnection object for other tasks (like writing to the URL) at the same time.

Sockets

URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet. Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application.

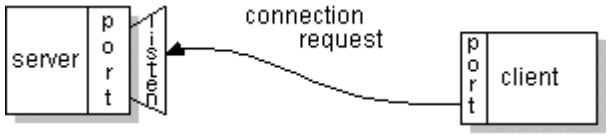
In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

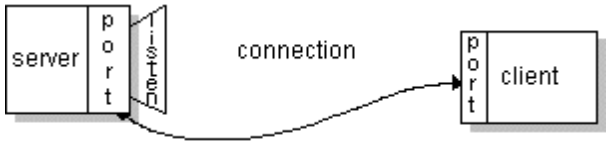
What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The **java.net package** in the Java platform provides a **class, Socket**, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, java.net includes the **ServerSocket** class, which implements a socket that **servers** can use to listen for and accept connections to clients.

If you are trying to connect to the Web, the URL class and related classes (URLConnection, URLEncoder) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation.

Establishing a Simple Server Using Stream Sockets

Establishing a simple server in Java requires five steps.

Step 1: Create a ServerSocket

First step is to create a **ServerSocket object**. A call to the ServerSocket constructor, such as

ServerSocket server = new ServerSocket(portNumber, queueLength);

registers an available TCP port number and specifies themaximum number of clients that can wait to connect to the server (i.e., the queue length). The port number is used by clients to locate the server application on the server computer. This is often called the handshake point. If the queue is full, the server refuses client connections. The constructor establishes the port where the server waits for connections from clients—a process known as **binding the server to the port**. Each client will ask to connect to the server on this port. Only one application at a time can be bound to a specific port on the server.

Step 2: Wait for a Connection

Programs manage each client connection with a Socket object. In Step 2, the server listens indefinitely (or blocks) for an attempt by a client to connect. To listen for a client connection, the program calls ServerSocket method accept, as in

Socket connection = server.accept();

which returns a Socket when a connection with a client is established. The Socket allows the server to interact with the client. The interactions with the client actually occur at a different server port from the handshake point. This allows the port specified in Step 1 to be used again in a multithreaded server to accept another client connection.

Step 3: Get the Socket's I/O Streams

Step 3 is to get the **OutputStream** and **InputStream** objects that enable the server to communicate with the client by sending and receiving bytes. The server sends information to the client via an OutputStream and receives information from the client via an InputStream. The server invokes method **getOutputStream** on the Socket to get a reference to the Socket's OutputStream and invokes method **getInputStream** on the Socket to get a reference to the Socket's InputStream.

```
Socket con=new Socket("localHost",95);
BufferedReader in=new BufferedReader(new InputStreamReader(con.getInputStream()));
PrintWriter out=new PrintWriter(con.getOutputStream(),true);
```

The beauty of establishing these relationships is that whatever the server writes to the PrintWriter is sent via the OutputStream and is available at the client's InputStream, and whatever the client writes to its OutputStream (with a corresponding PrintWriter) is available via the server's InputStream. The transmission of the data over the network is seamless and is handled completely by Java.

Step 4: Perform the Processing

In which the server and the client communicate via the OutputStream and InputStream objects.

Step 5: Close the Connection

when the transmission is complete, the server closes the connection by invoking the close method on the streams and on the Socket.

```
in.close();
out.close();
con.close();
```

Establishing a Simple Client Using Stream Sockets

Establishing a simple client in Java requires four steps.

Step 1: Create a Socket to Connect to the Server

In first step we create a Socket to connect to the server. The Socket constructor establishes the connection. For example, the statement

Socket connection = new Socket(serverAddress, port);

uses the Socket constructor with two arguments—**the server's address (serverAddress)** and the **port number**. If the connection attempt is successful, this statement returns a Socket. A connection attempt that fails throws an instance of a subclass of IOException, so many programs simply catch IOException. An UnknownHostException occurs specifically when the system is unable to resolve the server name specified in the call to the Socket constructor to a corresponding IP address.

Step 2: Get the Socket's I/O Streams

Here the client uses Socket methods getInputStream and getOutputStream to obtain references to the Socket's InputStream and OutputStream as described earlier.

Step 3: Perform the Processing

In this phase the client and the server communicate via the InputStream and OutputStream objects.

Step 4: Close the Connection

In Step 4, the client closes the connection when the transmission is complete by invoking the close method on the streams and on the Socket as described earlier.

InetAddress class

Usually, you don't have to worry too much about Internet addresses, the numerical host addresses that consist of four bytes (or, with IPv6, 16 bytes) such as 132.163.4.102. However, you can use the **InetAddress** class if you need to convert between host names and Internet addresses.

As of JDK 1.4, the **java.net package** supports IPv6 Internet addresses, provided the host operating system does.

The static **getByName** method returns an InetAddress object of a host. For example,

InetAddress address = InetAddress.getByName("HostName");

returns an InetAddress object that encapsulates the sequence of four bytes such as 132.163.4.104.

Some host names with a lot of traffic correspond to multiple Internet addresses, to facilitate load balancing. For example,the host name java.sun.com corresponds to three different Internet addresses. One of them is picked at random when the host is accessed. You can get all hosts with the **getAllByName** method.

InetAddress[] addresses = InetAddress.getAllByName(host);

String getHostAddress()-returns a string with decimal numbers, separated by periods, for example, "132.163.4.102".

String getHostName()-returns the host name.

Program for chatting between client and server

```
//Server.java
import java.io.*;
import java.net.*;
public class Server
```

```
{
public static void main(String a[])throws IOException
{
try
{
System.out.println("SERVER:.....\n");
ServerSocket s=new ServerSocket(95);
System.out.println("Server Waiting For The Client");
Socket cs=s.accept();
InetAddress ia=cs.getInetAddress();
String cli=ia.getHostAddress();
System.out.println("Connected to the client with IP:"+cli);
BufferedReader in=new BufferedReader(new
InputStreamReader(cs.getInputStream()));
PrintWriter out=new PrintWriter(cs.getOutputStream(),true);
do
{
BufferedReader din=new BufferedReader(new
InputStreamReader(System.in));
System.out.print("To Client:");
String tocl=din.readLine();
out.println(tocl);
String st=in.readLine();
if(st.equalsIgnoreCase("Bye")||st==null)break;
System.out.println("From Client:"+st);
}while(true);
in.close();
out.close();
cs.close();
}
catch(IOException e) { }
}
}
```

//Client.java

```
import java.io.*;
import java.net.*;
public class Client
{
public static void main(String a[])throws IOException
{
try
```

```
{
System.out.println("CLIENT:.....\n");
Socket con=new Socket("localHost",95);
BufferedReader in=new BufferedReader(new
InputStreamReader(con.getInputStream()));
PrintWriter out=new PrintWriter(con.getOutputStream(),true);
while(true)
{
String s1=in.readLine();
System.out.println("From Server:"+s1);
System.out.print("Enter the messages to the server:");
BufferedReader din=new BufferedReader(new
InputStreamReader(System.in));
String st=din.readLine();
out.println(st);
if(st.equalsIgnoreCase("Bye")||st==null)break;
}
in.close();
out.close();
con.close();
}
catch(UnknownHostException e){ }
}
}
```

Serving Multiple Clients

There is one problem with the simple server in the preceding example. Suppose we want to allow multiple clients to connect to our server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to use the server at the same time. Rejecting multiple connections allows any one client to monopolize the service by connecting to it for a long time. We can do much better through the magic of threads.

Every time we know the program has established a new socket connection, that is, when the call to accept was successful, we will launch a new thread to take care of the connection between the server and that client. The main program will just go back and wait for the next connection. For this to happen, the main loop of the server should look like this:

```
while (true)
{
  Socket incoming = s.accept();
  Runnable r = new ThreadedEchoHandler(incoming);
```

```
  Thread t = new Thread(r);
  t.start();
}
```

The THReadedEchoHandler class implements Runnable and contains the communication loop with the client in its run method.

```
class ThreadedEchoHandler implements Runnable
{ . . .
  public void run()
  {
    try
    {
      InputStream inStream = incoming.getInputStream();
      OutputStream outStream = incoming.getOutputStream();
      ...process input and send response...
      incoming.close();
    }
    catch(IOException e)
    {
      handle exception
    }
  }
}
```

Because each connection starts a new thread, multiple clients can connect to the server at the same time.

Socket Timeouts

In real-life programs, you don't just want to read from a socket, because the read methods will block until data are available. If the host is unreachable, then your application waits for a long time and you are at the mercy of the underlying operating system to time out eventually. Instead, you should decide what timeout value is reasonable for your particular application. Then, call the **setSoTimeout** method to set a timeout value (in milliseconds).

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // time out after 10 seconds
```

If the timeout value has been set for a socket, then all subsequent read and write operations throw a **SocketTimeoutException** when the timeout has been reached before the operation has completed its work. You can catch that exception and react to the timeout.

```
try
{
  Scanner in = new Scanner(s.getInputStream());
  String line = in.nextLine();
```

```
  . . .
}
catch (InterruptedIOException exception)
{
  react to timeout
}
```

Interruptible Sockets

When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed. Similarly, when you read or write data through a socket, the current thread blocks until the operation is successful or has timed out.

In interactive applications, you would like to give users an option to simply cancel a socket connection that does not appear to produce results. However, if a thread blocks on an unresponsive socket, you cannot unblock it by calling interrupt.

To interrupt a socket operation, you use a **SocketChannel**, a feature of the **java.nio package.** Open the SocketChannel like this:

SocketChannel channel = SocketChannel.open(new InetSocketAddress(host, port));

A channel does not have associated streams. Instead, it has read and write methods that make use of Buffer objects.These methods are declared in interfaces **ReadableByteChannel** and **WritableByteChannel**.

If you don't want to deal with buffers, you can use the Scanner class to read from a SocketChannel because Scanner has a constructor with a ReadableByteChannel parameter:

Scanner in = new Scanner(channel);

To turn a channel into an output stream, use the static Channels.newOutputStream method.

OutputStream outStream = Channels.newOutputStream(channel);

That's all you need to do. **Whenever a thread is interrupted during an open, read, or write operation, the operation does not block but is terminated with an exception.**

Half-Close

When a client program sends a request to the server, the server needs to be able to determine when the end of the request occurs. For that reason, many Internet protocols (such as SMTP) are line oriented. Other protocols contain a header that specifies the size of the request data. Otherwise, indicating the end of the request data is harder than writing data to a file. With a file, you'd just close the file at the end of the data. However, if you close a socket, then you immediately disconnect from the server.

The half-close overcomes this problem. You can close the output stream of a socket, thereby indicating to the server the end of the request data, but keep the input stream open so that you can read the response.

The client side looks like this:

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// send request data
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// now socket is half closed
// read response data
while (in.hasNextLine()) != null) { String line = in.nextLine(); . . . }
socket.close();
```

The server side simply reads input until the end of the input stream is reached.This protocol is only useful for one-shot services such as HTTP where the client connects, issues a request, catches the response, and then disconnects.

Sending E-Mail via javax.mail API

```
//SendMail.java
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendMail {

	public static void main(String[] args) {

		final String username = "knavin12@gmail.com";
		final String password = "password";

		Properties props = new Properties();
		props.put("mail.smtp.auth", "true");
		props.put("mail.smtp.starttls.enable", "true");
		props.put("mail.smtp.host", "smtp.gmail.com");
		props.put("mail.smtp.port", "587");
```

```
		Session session = Session.getInstance(props,
		  new javax.mail.Authenticator() {
			protected PasswordAuthentication getPasswordAuthentication() {
				return new PasswordAuthentication(username, password);
			}
		  });

		try {

			Message message = new MimeMessage(session);
			message.setFrom(new InternetAddress("knavin12@gmail.com"));
			message.setRecipients(Message.RecipientType.TO,
				InternetAddress.parse("knavin12@gmail.com"));
			message.setSubject("Testing Subject");
			message.setText("Hello Navin,"
				+ "\n\n Congrates u succeded sending mail\n through Java.mail
				API.");

			Transport.send(message);

			System.out.println("Your email has been sent successfully");

		} catch (MessagingException e) {
			throw new RuntimeException(e);
		}
	}
}
```

Chapter -6 Java Beans

A **Java Bean** is a software component that has been designed to be **reusable** in a variety of different environments. There is no restriction on the capability of a Bean. It may perform a simple function, such as checking the spelling of a document, or a complex function, such as forecasting the performance of a stock portfolio.

Beans are important, because they allow you to build complex systems from software components. These components may be provided by you or supplied by one or more different vendors. Java Beans defines an architecture that specifies how these building blocks can operate together. To better understand the value of Beans, consider the following. Hardware designers
have a wide variety of components that can be integrated together to construct a system. Resistors, capacitors, and inductors are examples of simple building blocks. Integrated circuits provide more advanced functionality. All of these different parts can be reused. It is not necessary or possible to rebuild these capabilities each time a new system is needed. Also, the same pieces can be used in different types of circuits. This is possible because the behavior of these components is understood and documented.

Unfortunately, the software industry has not been as successful in achieving the benefits of reusability and interoperability. Large applications grow in complexity and become very difficult to maintain and enhance. Part of the problem is that, until recently, there has not been a standard, portable way to write a software component. To achieve the benefits of component software, a component architecture is needed that allows programs to be assembled from software building blocks, perhaps provided by different vendors. It must also be possible for a designer to select a component, understand its capabilities, and incorporate it into an application. When a new version of a component becomes available, it should be easy to incorporate this functionality into existing code.Fortunately, Java Beans provides just such an architecture.

Advantages of Java Beans

A software component architecture provides standard mechanisms to deal with software building blocks. The following list enumerates some of the specific benefits that Java technology provides for a component developer:

- Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

Steps for creating a new Bean using netbeans

Here are the steps that you must follow to create a new Bean:

1. Create a directory for the new Bean.

2. Create the Java source file(s).

3. Compile the source file(s).

4. Create a manifest file.

5. Generate a JAR file.

6. Start the BDK (Bean Development Kit).

7. Test.

Explanation

1. Create a Directory for the New Bean (You can use netbeans and create a new project for that)

2. Create the Source File for the New Bean

```
//Colors.java
import java.awt.*;
import java.awt.event.*;

public class Colors extends Canvas {
transient private Color color;
private boolean rectangular;
public Colors() {
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent me) {
change();
}
});
rectangular = false;
setSize(200, 100);
change();
}
public boolean getRectangular() {
return rectangular;
}
public void setRectangular(boolean flag) {
this.rectangular = flag;
```

```
repaint();
}
public void change() {
     color = randomColor();
repaint();
}
private Color randomColor() {
int r = (int)(255*Math.random());
int g = (int)(255*Math.random());
int b = (int)(255*Math.random());
return new Color(r, g, b);
}
public void paint(Graphics g) {
Dimension d = getSize();
int h = d.height;
int w = d.width;
g.setColor(color);
if(rectangular) {
g.fillRect(0, 0, w-1, h-1);
}
else {
g.fillOval(0, 0, w-1, h-1);
}
}
}
```

3. Compile the Source Code for the New Bean

Compile the source code to create a class file. Type the following:
javac Colors.java.

4. Create a Manifest File

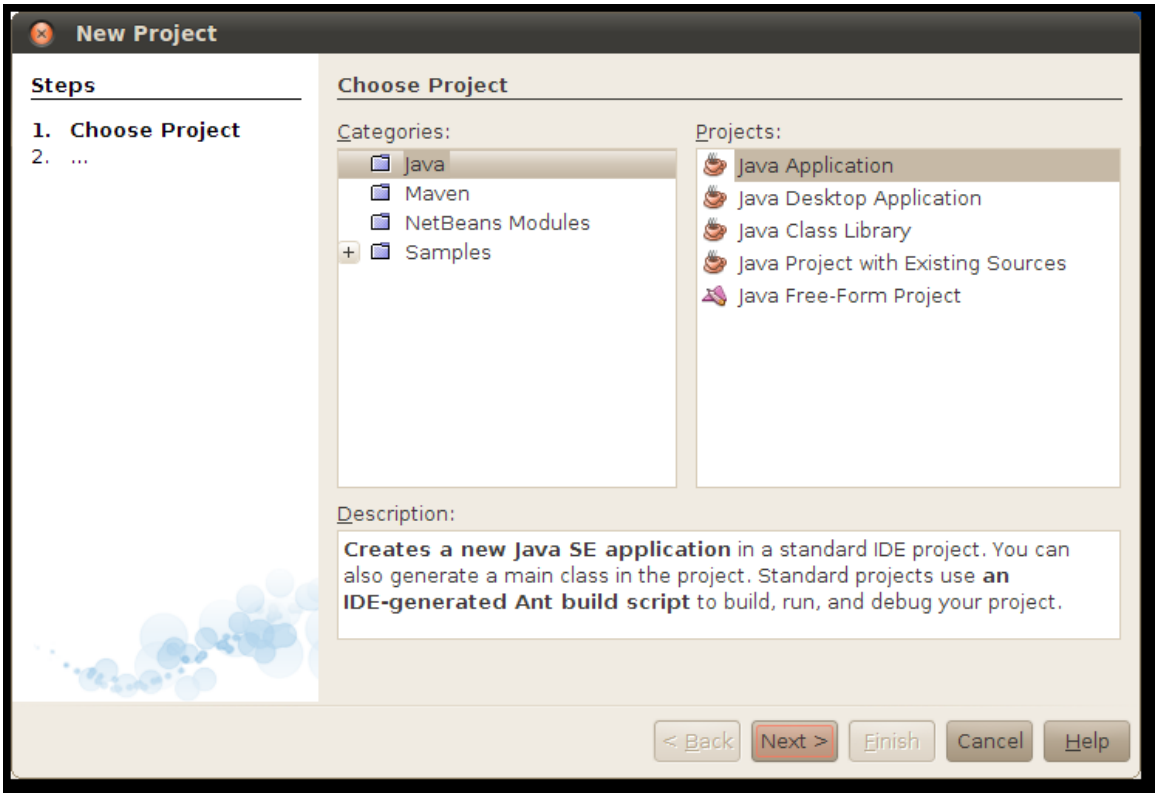
You must now create a manifest file. First, switch to the c:\bdk\demo directory. This is the directory in which the manifest files for the BDK demos are located. Put the source code for your manifest file in the file colors.mft. It is shown here:

```
Name: sunw/demo/colors/Colors.class
Java-Bean: True
```

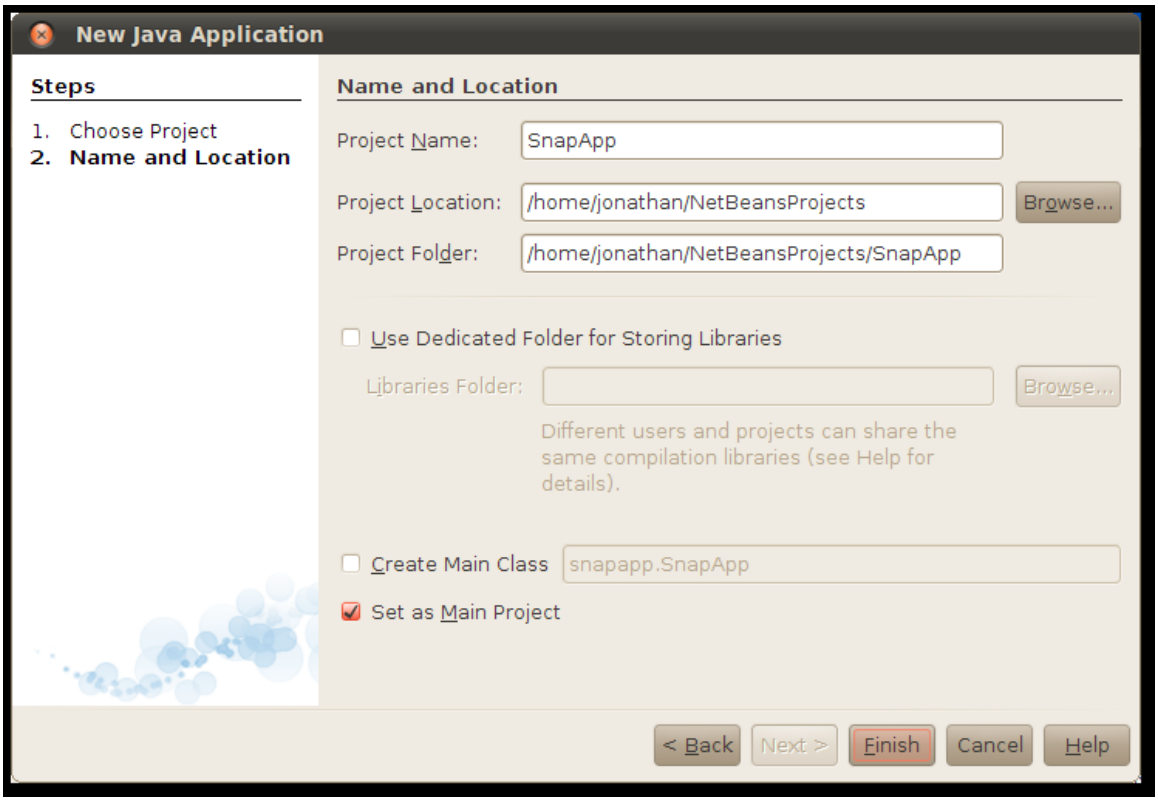
This file indicates that there is one .class file in the JAR file and that it is a Java Bean. Notice that the Colors.class file is in the package sunw.demo.colors and in the subdirectory sunw\demo\colors relative to the current directory.

Creating Java Beans using NetBeans

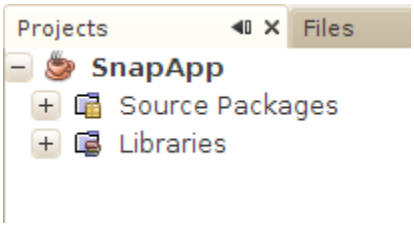
1. Start NetBeans. Choose File > New Project... from the menu.



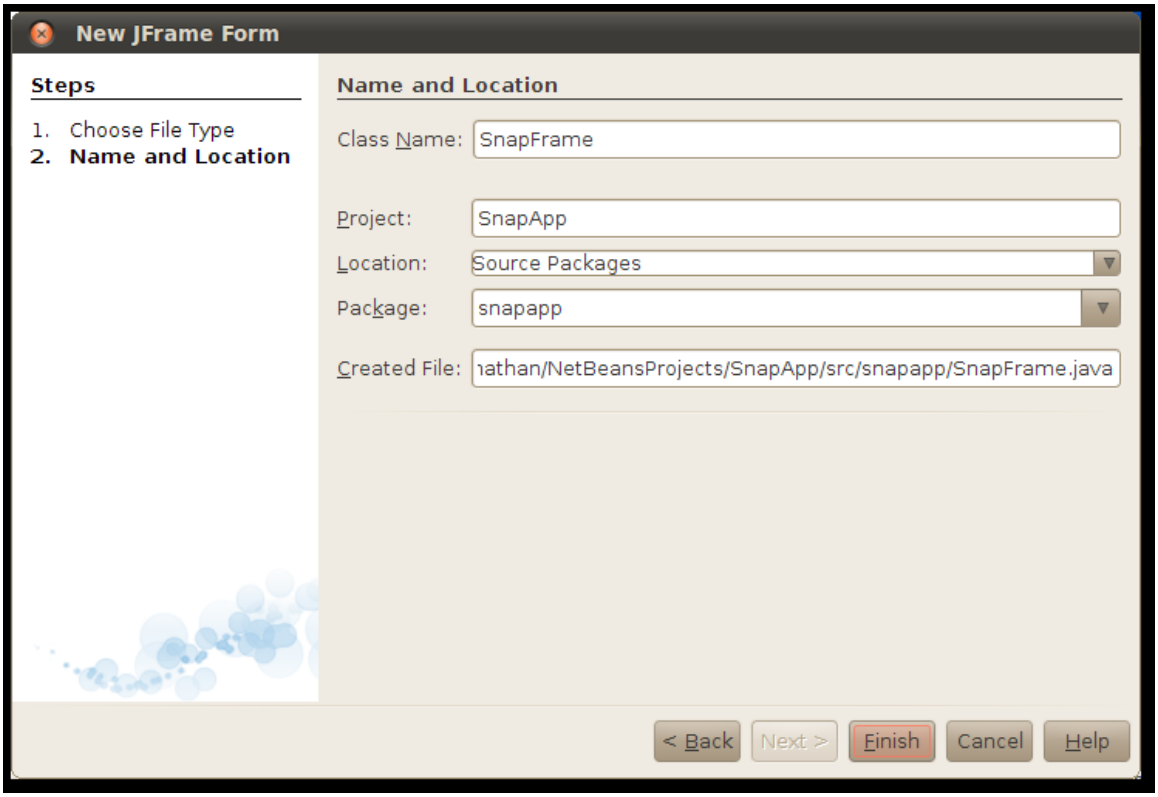
2. Select Java from the Categories list and select Java Application from the Projects list. Click Next



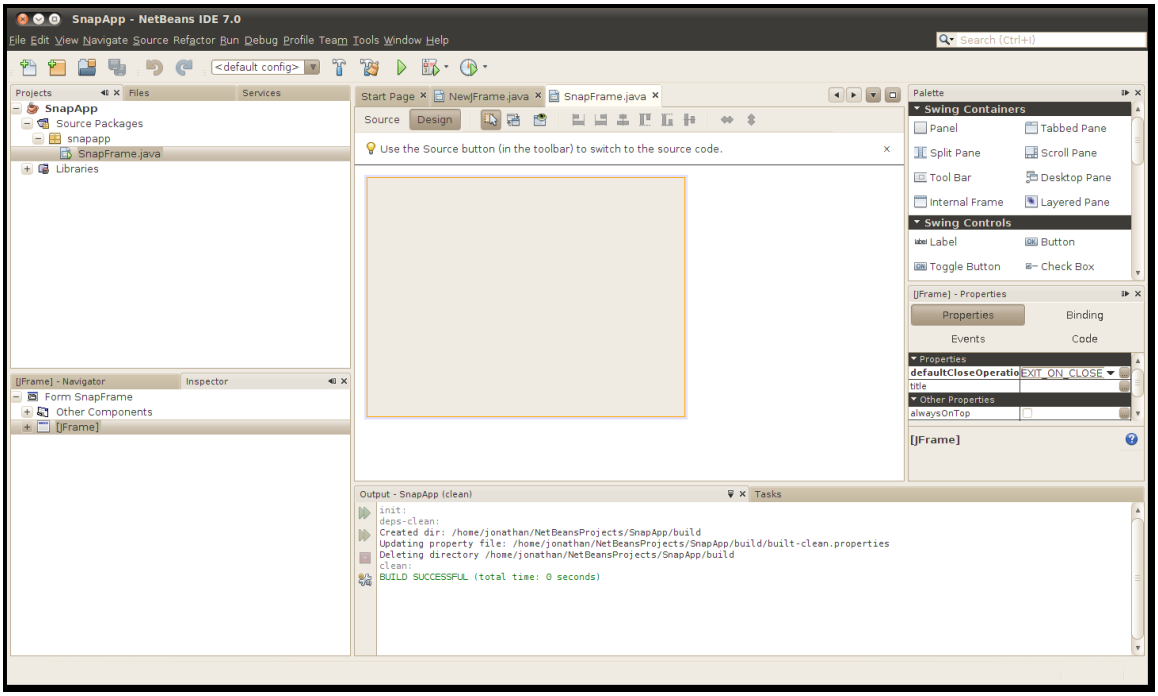
3. Enter SnapApp as the application name. Uncheck Create Main Class and click Finish. NetBeans creates the new project and you can see it in NetBeans' Projects pane:



4. Right-click on the SnapApp project and choose New > JFrame Form... from the popup menu.



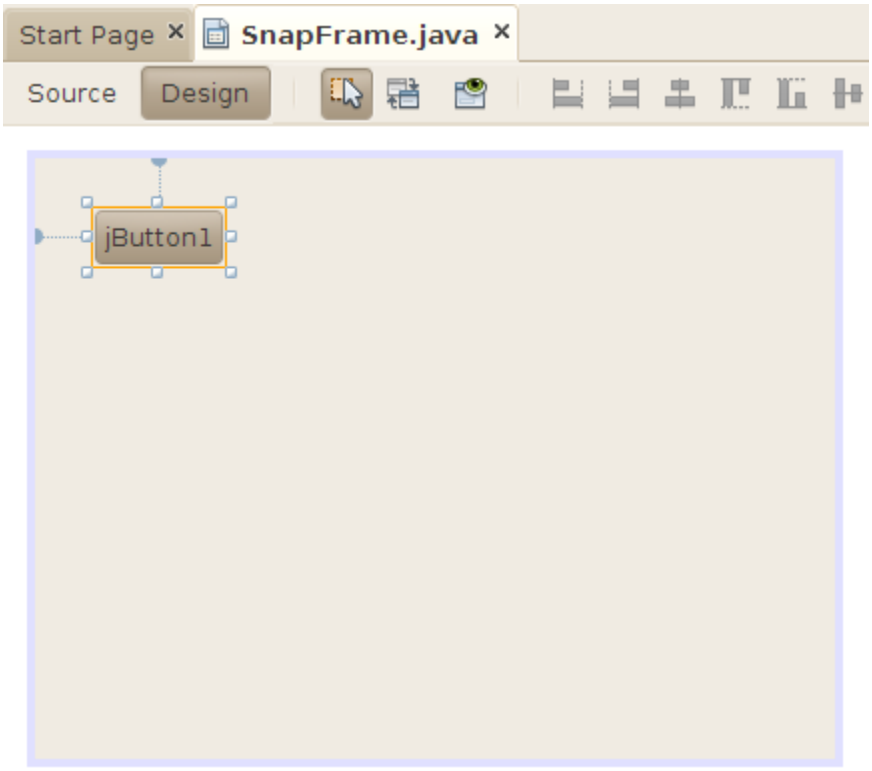
5. Fill in SnapFrame for the class name and snapapp as the package. Click Finish. NetBeans creates the new class and shows its visual designer:



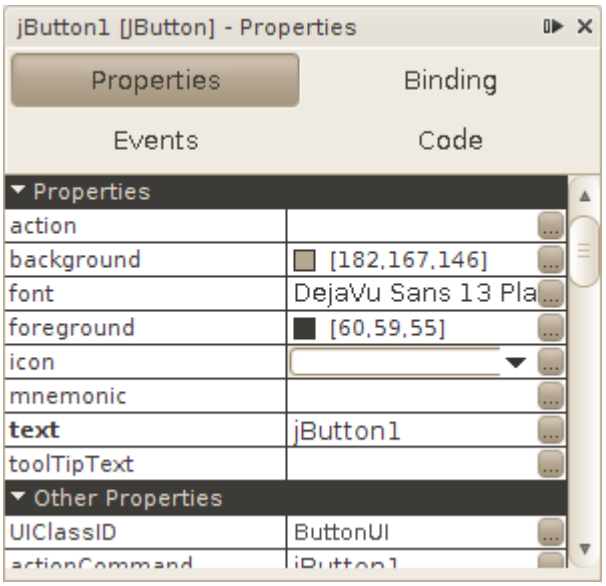
In the Projects pane on the left, you can see the newly created SnapFrame class. In the center of the screen is the NetBeans visual designer. On the right side is the Palette, which contains all the

components you can add to the frame in the visual designer.

6. Take a closer look at the Palette. All of the components listed are beans. The components are grouped by function. Scroll to find the Swing Controls group, then click on Button and drag it over into the visual designer. The button is a bean!



Under the palette on the right side of NetBeans is an inspector pane that you can use to examine and manipulate the button. Try closing the output window at the bottom to give the inspector pane more space.



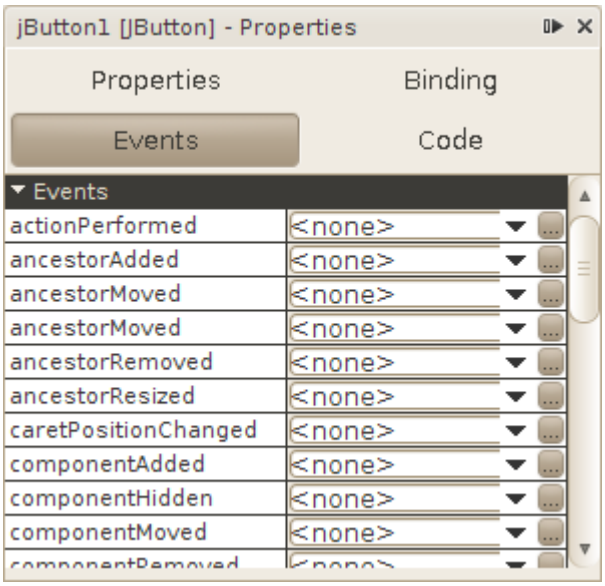
Properties

The properties of a bean are the things you can change that affect its appearance or internal state. For the button in this example, the properties include the foreground color, the font, and the text that appears on the button. The properties are shown in two groups. Properties lists the most frequently used properties, while Other Properties shows less commonly used properties.

Go ahead and edit the button's properties. For some properties, you can type values directly into the table. For others, click on the ... button to edit the value. For example, click on ... to the right of the foreground property. A color chooser dialog pops up and you can choose a new color for the foreground text on the button. Try some other properties to see what happens. Notice you are not writing any code.

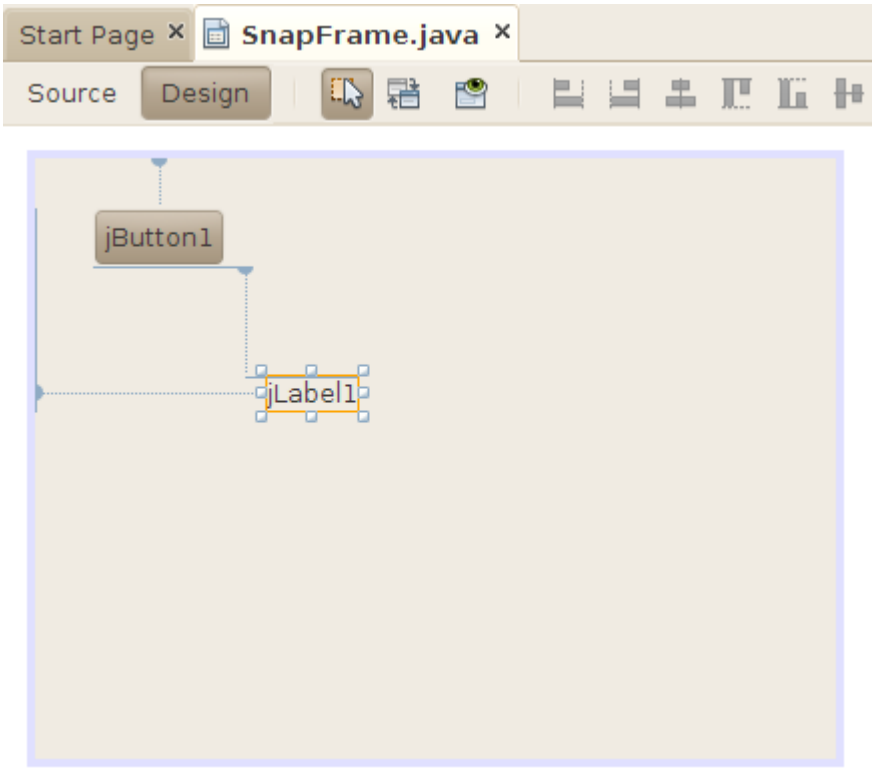
Events

Beans can also fire events. Click on the Events button in the bean properties pane. You'll see a list of every event that the button is capable of firing.



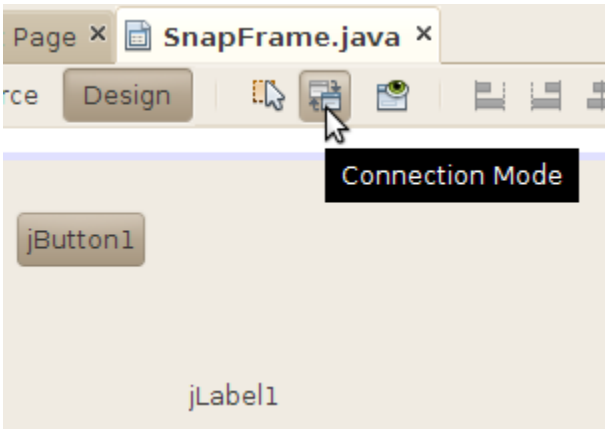
You can use NetBeans to hook up beans using their events and properties. To see how this works, drag a Label out of the palette into the visual designer for SnapFrame.

Add a label to the visual designer

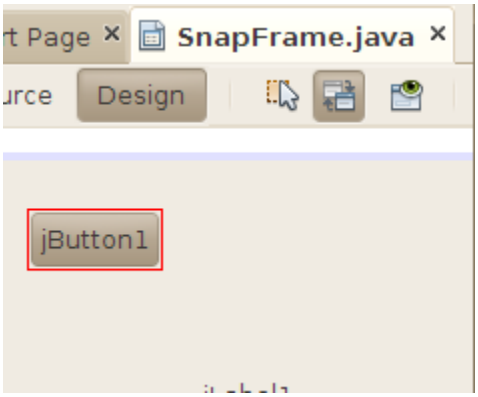


Wiring the Application

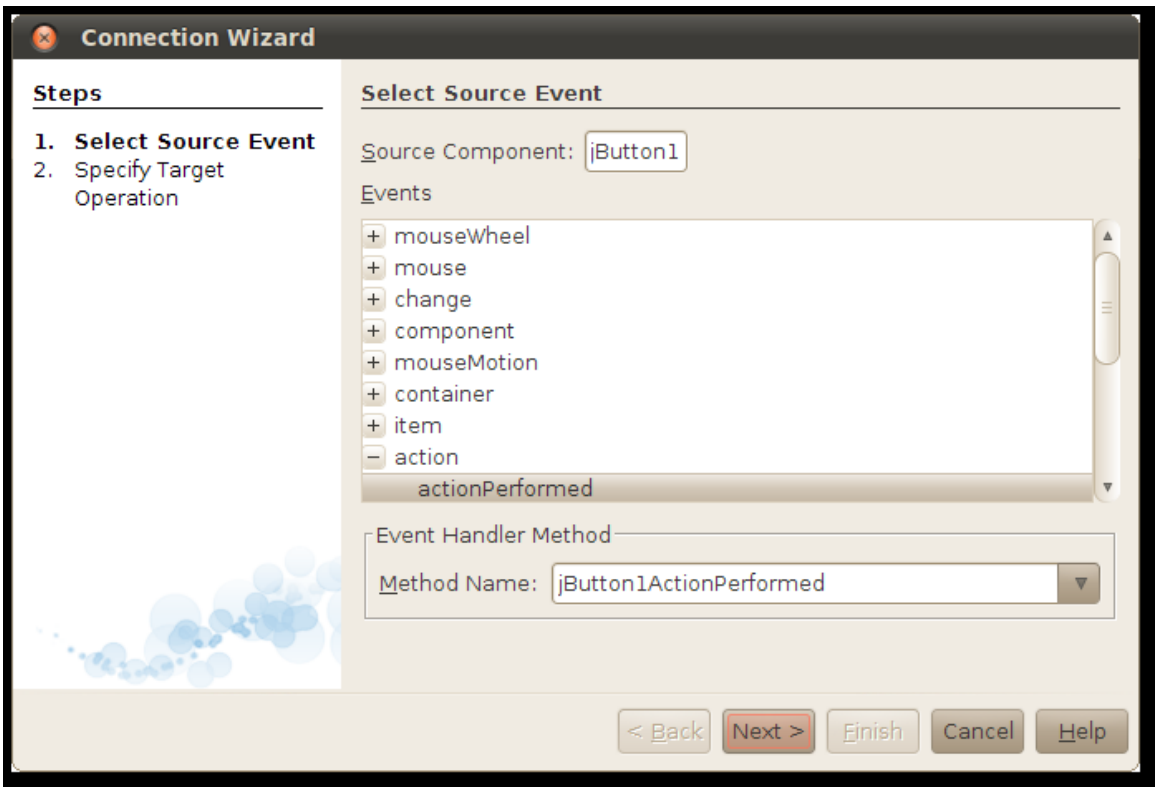
To wire the button and the label together, click on the Connection Mode button in the visual designer toolbar.



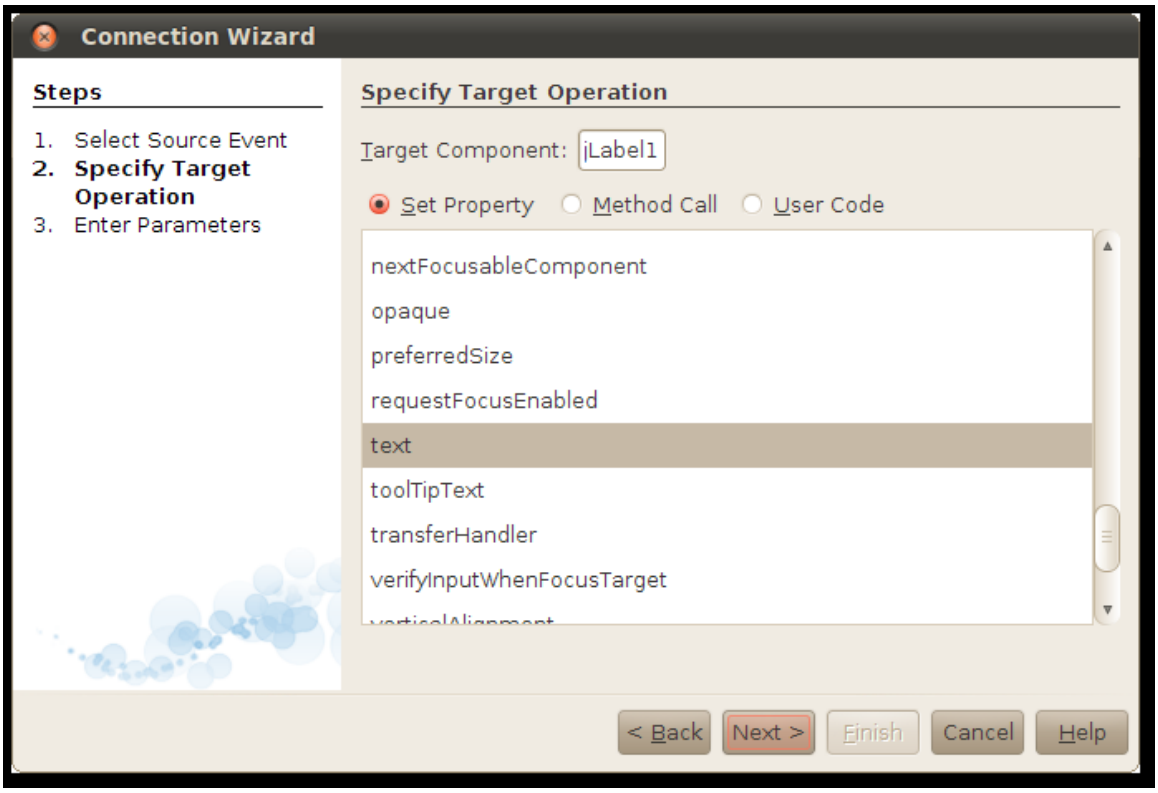
Click on the button in the SnapFrame form. NetBeans outlines the button in red to show that it is the component that will be generating an event.



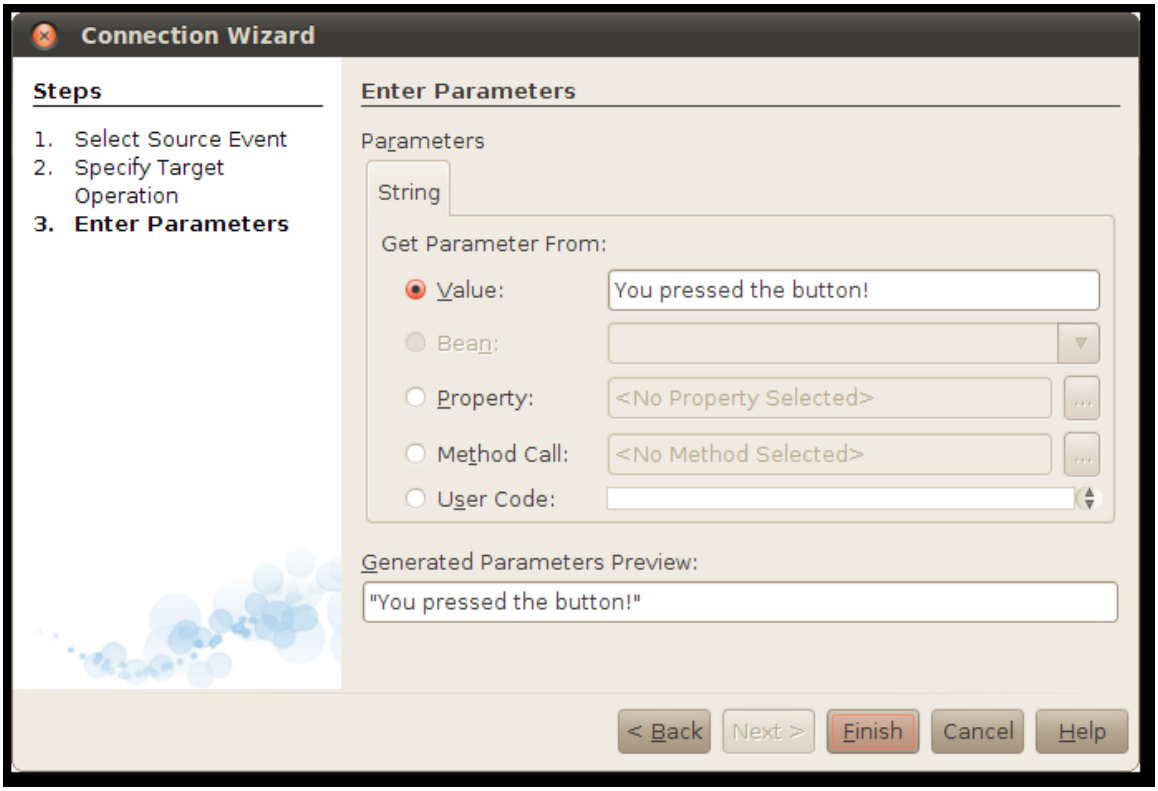
Click on the label. NetBeans' Connection Wizard pops up. First you will choose the event you wish to respond to. For the button, this is the action event. Click on the + next to action and select actionPerformed. Click Next >.



Now you get to choose what happens when the button fires its action event. The Connection Wizard lists all the properites in the label bean. Select text in the list and click Next.



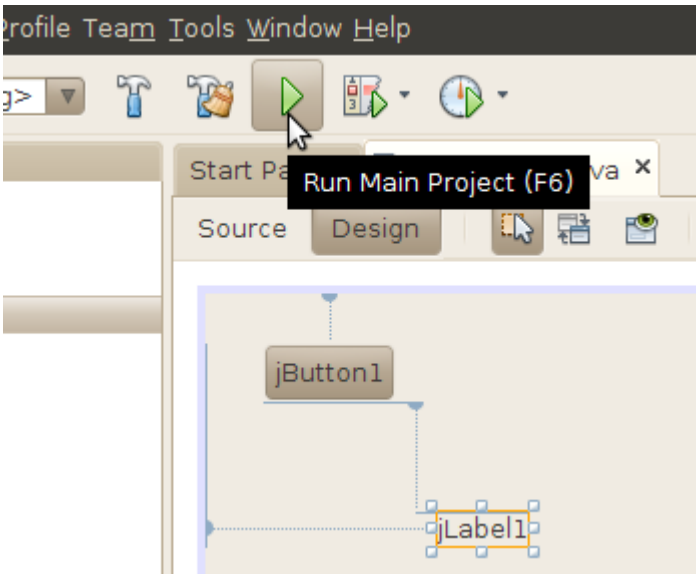
In the final screen of the Connection Wizard, fill in the value you wish to set for the text property. Click on Value, then type You pressed the button! or something just as eloquent. Click Finish.



NetBeans wires the components together and shows you its handiwork in the source code editor.

```
    @SuppressWarnings("unchecked")
    Generated Code

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        jLabel1.setText("You pressed the button!");
    }

    /**
```

Click on the Design button in the source code toolbar to return to the UI designer. Click Run Main Project or press F6 to build and run your project.



NetBeans builds and runs the project. It asks you to identify the main class, which is SnapFrame. When the application window pops up, click on the button. You'll see your immortal prose in the label.

Notice that you did not write any code. This is the real power of JavaBeans — with a good builder tool like NetBeans, you can quickly wire together components to create a running application.

Using a Third-Party Bean

Almost any code can be packaged as a bean. The beans you have seen so far are all visual beans, but beans can provide functionality without having a visible component.

The power of JavaBeans is that you can use software components without having to write them or understand their implementation.

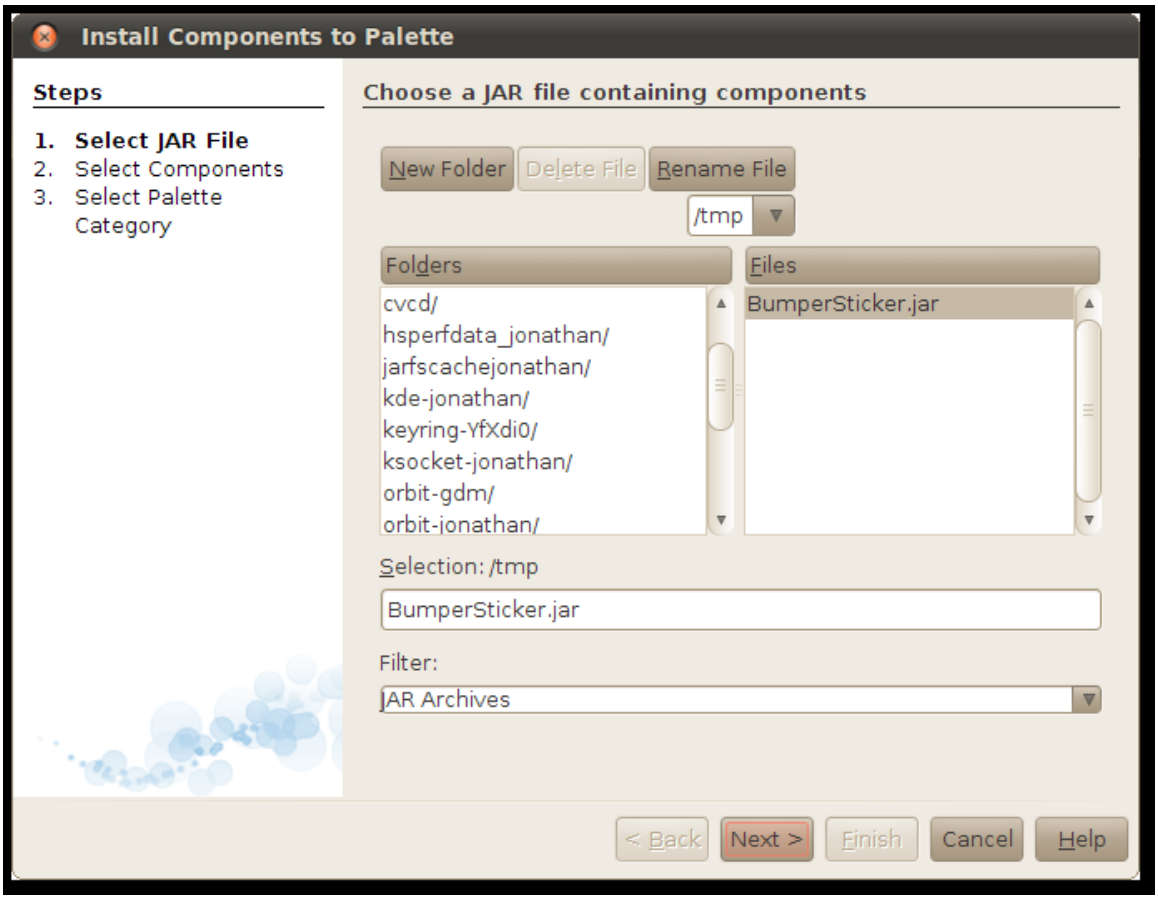
This page describes how you can add a JavaBean to your application and take advantage of its functionality.

Adding a Bean to the NetBeans Palette

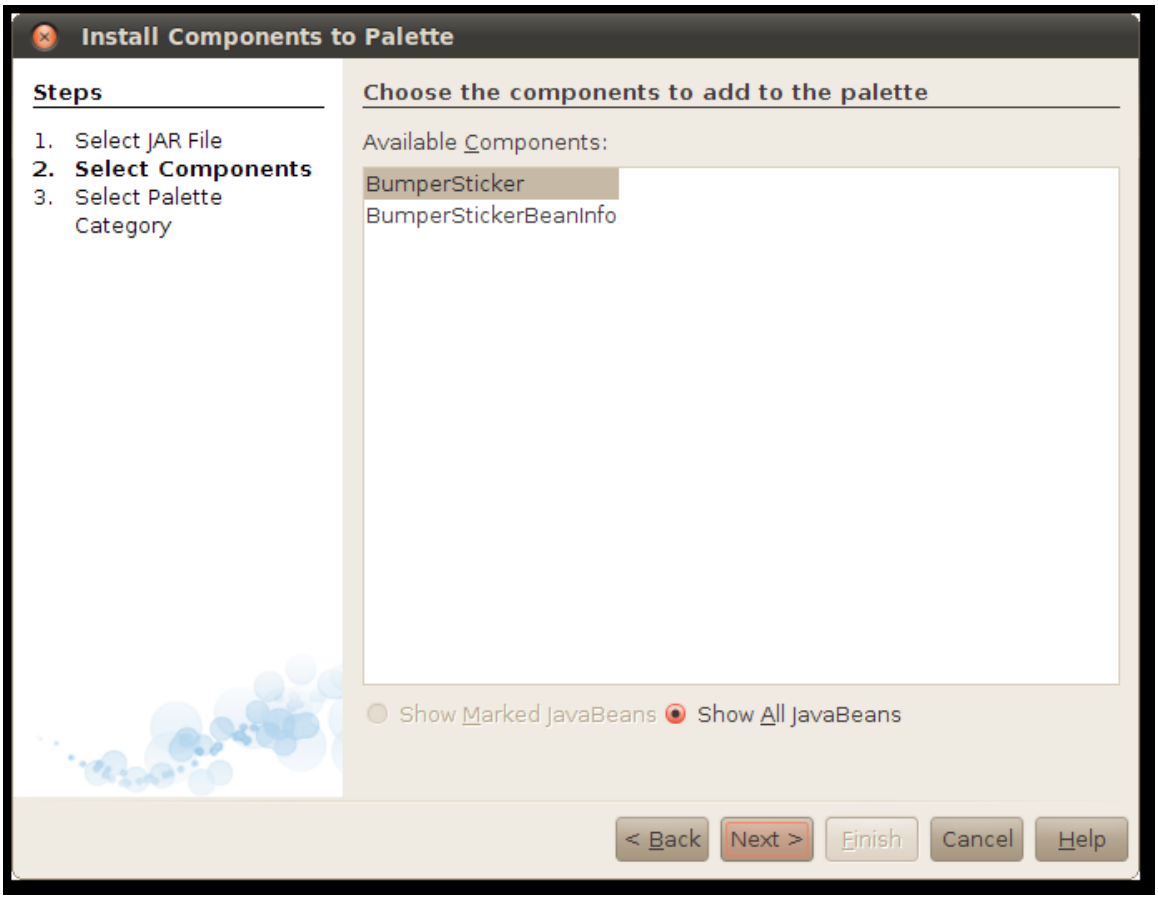
Download an example JavaBean component, BumperSticker. Beans are distributed as JAR files. Save the file somewhere on your computer. BumperSticker is graphic component and exposes one method, go(), that kicks off an animation.

To add BumperSticker to the NetBeans palette, choose Tools > Palette > Swing/AWT Components from the NetBeans menu.

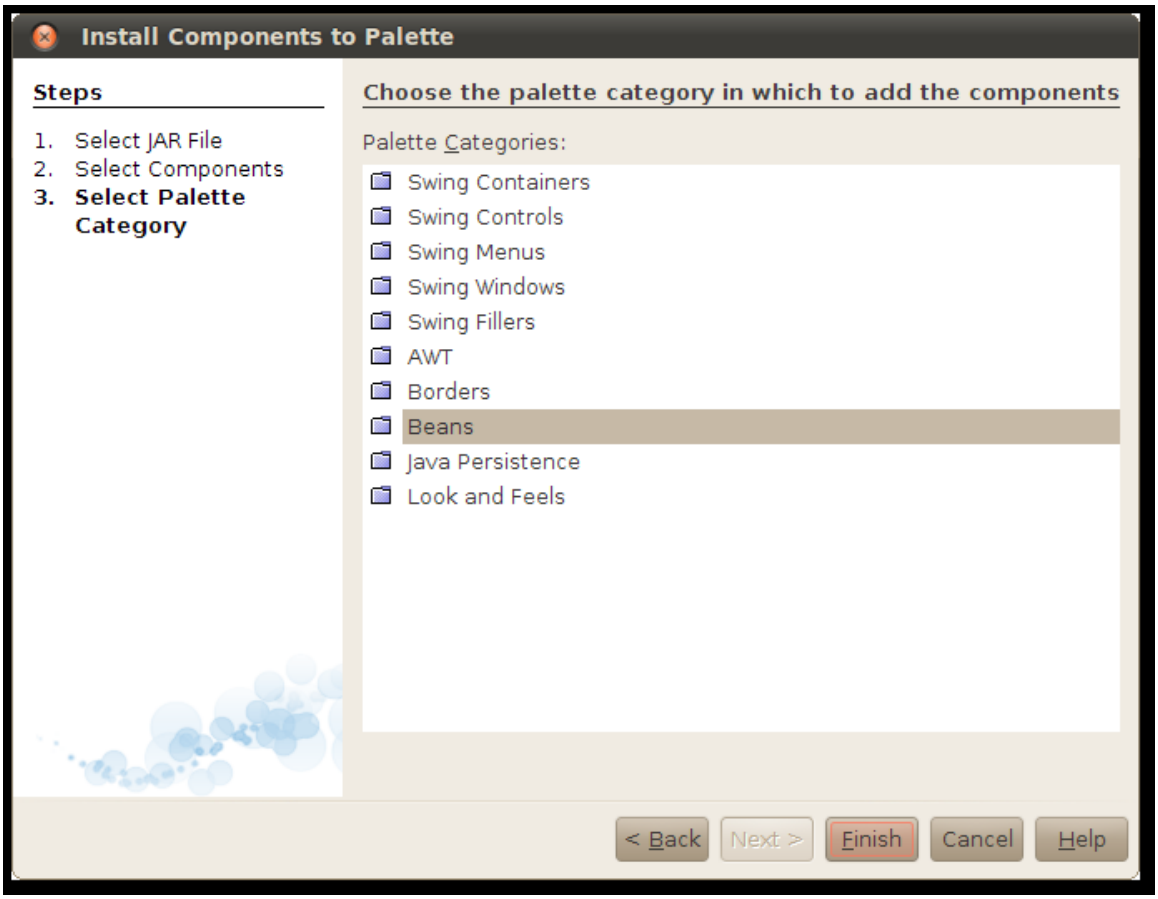
Click on the Add from JAR... button. NetBeans asks you to locate the JAR file that contains the beans you wish to add to the palette. Locate the file you just downloaded and click Next.



NetBeans shows a list of the classes in the JAR file. Choose the ones you wish you add to the palette. In this case, select BumperSticker and click Next



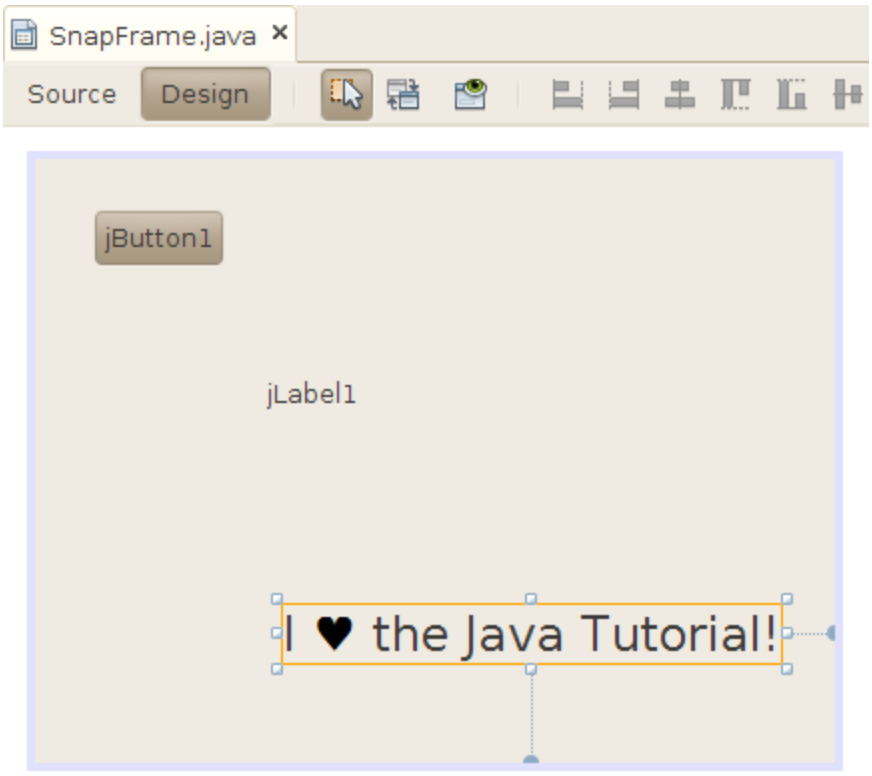
Finally, NetBeans needs to know which section of the palette will receive the new beans. Choose Beans and click Finish.



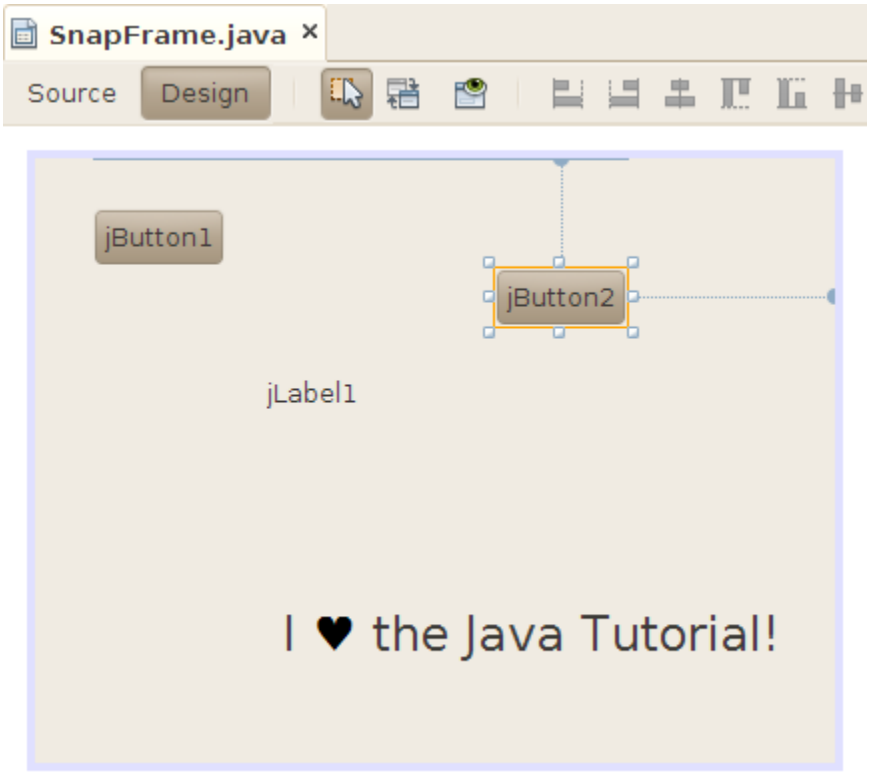
Click Close to make the Palette Manager window go away. Now take a look in the palette. BumperSticker is there in the Beans section.

Using Your New JavaBean

Go ahead and drag BumperSticker out of the palette and into your form.



You can work with the BumperSticker instance just as you would work with any other bean. To see this in action, drag another button out into the form. This button will kick off the BumperSticker's animation.



Wire the button to the BumperSticker bean, just as you already wired the first button to the text field.

Begin by clicking on the Connection Mode button.

Click on the second button. NetBeans gives it a red outline.

Click on the BumperSticker component. The Connection Wizard pops up.

Click on the + next to action and select actionPerformed. Click Next >.

Select Method Call, then select go() from the list. Click Finish.

Run the application again. When you click on the second button, the BumperSticker component animates the color of the heart.

Again, notice how you have produced a functioning application without writing any code.

Unit 7. Servlets and Java Server Pages

Servlets

Servlets are small programs that execute on the server side of a Web connection. Just as applets dynamically extend the functionality of a Web browser, servlets dynamically extend the functionality of a Web server.

A servlet is a Java programming language class used to extend the capabilities of servers that host applications accessed via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by Web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.The **javax.servlet** and **javax.servlet.http** packages provide interfaces and classes for writing servlets. All servlets must implement the **Servlet** interface, which defines life-cycle methods.

The Life Cycle of a Servlet

Three methods are central to the life cycle of a servlet. These are **init( ), service( ),** and **destroy( ).** They are implemented by every servlet and are invoked at specific times by the server. Let us consider a typical user scenario to understand when these methods are called.

First, when a user enters a **Uniform Resource Locator (URL)** to a Web browser. The **browser then generates an HTTP request for this URL.** This request is then sent to the appropriate server.

Second, this HTTP request is received by the Web server. The server maps this request to a particular servlet. The servlet is dynamically retrieved and loaded into the address space of the server.

Third, the **server** invokes **the init( ) method** of the **servlet**. **This method is invoked only when the servlet is first loaded into memory.** It is possible to pass initialization parameters to the servlet so it may configure itself.

Fourth, the **server** invokes the **service( ) method** of the **servlet**. This **method is called to process the HTTP request**. It is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an **HTTP response** for the client. The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The service( ) method is called for each HTTP request.

Finally, the server may decide to **unload the servlet from its memory**. The server calls the **destroy( ) method** to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.

The Servlet API

Two packages contain the classes and interfaces that are required to build servlets. These are **javax.servlet** and **javax.servlet.http**. They constitute the Servlet API.These packages are not part of the Java core packages. Instead, they are standard extensions. Therefore, they are not included in the Java Software Development Kit. You must download Tomcat or Glass Fish server to obtain their functionality.

The javax.servlet Package

The **javax.servlet package** contains a number of interfaces and classes that **establish the framework in which servlets operate.**

The following table summarizes the **core interfaces** that are provided in this package. The most significant of these is **Servlet**. All servlets must implement this interface or extend a class that implements the interface.

The **ServletRequest** and **ServletResponse** interfaces are also very important.

Interface	**Description**
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletContext	Enables servlets to log events and access information about their environment.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.
SingleThreadModel	Indicates that the servlet is thread safe.

The following table summarizes the **core classes** that are provided in the javax.servlet package.

Class	**Description**
GenericServlet	Implements the Servlet and ServletConfig interfaces.
ServletInputStream	Provides an input stream for reading requests from a client.
ServletOutputStream	Provides an output stream for writing responses to a client.
ServletException	Indicates a servlet error occurred.
UnavailableException	Indicates a servlet is unavailable.

The Servlet Interface

All servlets must implement the **Servlet interface**. It declares the **init( ), service( ), and destroy( ) methods** that are called by the server during the life cycle of a servlet. The methods defined by Servlet are shown below:

Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig *sc*) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from *sc*. An **UnavailableException** should be thrown if the servlet cannot be initialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from *req*. The response to the client can be written to *res*. An exception is generated if a servlet or IO problem occurs.

Table 27-1. *The Methods Defined by* Servlet

The ServletRequest Interface

The ServletRequest interface is implemented by the server. It enables a servlet to obtain information about a client request. Several of its methods are summarized in Table below.

Method	Description
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value –1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A **null** value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream** that can be used to read binary data from the request. An **IllegalStateException** is thrown if **getReader( )** has already been invoked for this request.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String *name*)	Returns an array containing values associated with the parameter specified by *name*.
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStream( )** has already been invoked for this request.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, "http", "ftp").
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

Table 27-3. *Various Methods Defined by* ServletRequest

The ServletResponse Interface

The ServletResponse interface is implemented by the server. It enables a servlet to formulate a response for a client. Several of its methods are summarized in Table below.

Method	Description
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has already been invoked for this request.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has already been invoked for this request.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

Table 27-4. *Various Methods Defined by* ServletResponse

Note: For detailed information about **javax.servlet** package refer to the following link
http://docs.oracle.com/javaee/1.4/api/javax/servlet/package-summary.html

Reading Servlet Parameters

The **ServletRequest class** includes methods that allow to read the **names and values of parameters** that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A **Web page** is defined in **index.jsp** and a servlet is defined in **PostParametersServlet.java**. The HTML source code for index.jsp is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
//index.jsp
<html>
<body>
<center>
```

```
<form name="Form1" method="post" action="PostParametersServlet">
<table>
<tr>
<td><B>Employee</td>
<td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
<td><B>Phone</td>
<td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>

//PostParametersServlet.java
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet extends GenericServlet {

   public void service(ServletRequest request,ServletResponse response)
  throws ServletException, IOException {
// Get print writer.
PrintWriter pw = response.getWriter();
// Get enumeration of parameter names.
Enumeration e = request.getParameterNames();
// Display parameter names and values.
while(e.hasMoreElements()) {
String pname = (String)e.nextElement();
pw.print(pname + " = ");
String pvalue = request.getParameter(pname);
pw.println(pvalue);
}
pw.close();
}
}
```

output

```
e = navin
p = 9841
```

The javax.servlet.http Package

The **javax.servlet.http** package contains a number of interfaces and classes that are commonly used by **servlet developers**. You will see that its functionality makes it easy to build servlets that work with HTTP requests and responses.

The following table summarizes the core **interfaces** that are provided in this package:

Interface	**Description**
HttpServletRequest	Enables servlets to read data from an HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and written.
HttpSessionBindingListener	Informs an object that it is bound to or unbound from a session.

The following table summarizes the core **classes** that are provided in this package. The most important of these is HttpServlet. Servlet developers typically extend this class in order to process HTTP requests.

Class	**Description**
Cookie	Allows state information to be stored on a client machine.
HttpServlet	Provides methods to handle HTTP requests and responses.
HttpSessionEvent	Encapsulates a session-changed event.
HttpSessionBindingEvent	Indicates when a listener is bound to or unbound from a session value, or that a session attribute changed.

The HttpServletRequest Interface

The HttpServletRequest interface is implemented by the server. It enables a servlet to obtain information about a client request. Several of its methods are shown in Table below.

Method	**Description**
String getAuthType( )	Returns authentication scheme.
Cookie[ ] getCookies( )	Returns an array of the cookies in this request.
long getDateHeader(String *field*)	Returns the value of the date header field named *field*.
String getHeader(String *field*)	Returns the value of the header field named *field*.
Enumeration getHeaderNames( )	Returns an enumeration of the header names.
int getIntHeader(String *field*)	Returns the **int** equivalent of the header field named *field*.

String getMethod( )	Returns the HTTP method for this request.
String getPathInfo( )	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated( )	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString( )	Returns any query string in the URL.
String getRemoteUser( )	Returns the name of the user who issued this request.
String getRequestedSessionId( )	Returns the ID of the session.
String getRequestURI( )	Returns the URI.
StringBuffer getRequestURL( )	Returns the URL.
String getServletPath( )	Returns that part of the URL that identifies the servlet.
HttpSession getSession( )	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean *new*)	If *new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie( )	Returns **true** if a cookie contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdFromURL( )	Returns **true** if the URL contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdValid( )	Returns **true** if the requested session ID is valid in the current session context.

Table 27-5. *Various Methods Defined by* HttpServletRequest

The HttpServletResponse Interface

The HttpServletResponse interface is implemented by the server. It enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, SC_OK indicates that the HTTP request succeeded and SC_NOT_FOUND indicates that the requested resource is not available. Several methods of this interface are summarized in Table below.

Method	Description
void addCookie(Cookie *cookie*)	Adds *cookie* to the HTTP response.
boolean containsHeader(String *field*)	Returns **true** if the HTTP response header contains a field named *field*.
String encodeURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method.
void sendError(int *c*) throws IOException	Sends the error code *c* to the client.
void sendError(int *c*, String *s*) throws IOException	Sends the error code *c* and message *s* to the client.
void sendRedirect(String *url*) throws IOException	Redirects the client to *url*.
void setDateHeader(String *field*, long *msec*)	Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String *field*, String *value*)	Adds *field* to the header with value equal to *value*.
void setIntHeader(String *field*, int *value*)	Adds *field* to the header with value equal to *value*.
void setStatus(int *code*)	Sets the status code for this response to *code*.

Table 27-6. *Various Methods Defined by* HttpServletResponse

The Cookie Class

The Cookie class encapsulates a cookie. A cookie is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user's name, address, and other information. The user does not need to enter this data each time he or she visits the store. A servlet can write a cookie to a user's machine via the addCookie( ) method of the HttpServletResponse interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

The names and values of cookies are stored on the user's machine. Some of the information that is

saved for each cookie includes the following:

- The name of the cookie
- The value of the cookie
- The expiration date of the cookie
- The domain and path of the cookie

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends. Otherwise, the cookie is saved in a file on the user's machine.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the Web server. Otherwise, it is not.

The methods of the Cookie class are summarized in Table below

Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie must be sent using only a secure protocol. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the cookie protocol version. (Will be 0 or 1.)
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted. Passing –1 causes the cookie to be removed when the browser is terminated.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*, which means that cookies will be sent only when a secure protocol is being used.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int v)	Sets the cookie protocol version to *v*, which will be 0 or 1.

Table 27-8. *The Methods Defined by* Cookie

The HttpServlet Class

The HttpServlet class extends GenericServlet. It is commonly used when developing servlets that receive

and process HTTP requests. The methods of the HttpServlet class are summarized in Table below.

Method	Description
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP DELETE.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP GET.
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP HEAD.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP OPTIONS.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP POST.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP PUT.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Performs an HTTP TRACE.
long getLastModified(HttpServletRequest *req*)	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Table 27-9. *The Methods Defined by* HttpServlet

Handling HTTP Requests and Responses

The HttpServlet class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete( ), doGet( ), doHead( ), doOptions( ), doPost( ), doPut( ), and doTrace( ).**

Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a Web page is submitted.

//index.jsp

<html>

```
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Testing GET</title>
  </head>
  <body>
    <form action="testingget" method="get">
      <label style="color: green;"> <b>Testing Get:</b></label><br/></br>
      First Name: <input type="text" name="firstName" size="20"><br /><br/>
      Last Name:  <input type="text" name="surname" size="20">
    <br /><br />
    <input type="submit" value="Submit"></br></br>
  </form>
</body>
</html>

//TestingGet
import java.io.PrintWriter;
import java.io.IOException;
import java.sql.*;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.UnavailableException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class TestingGet extends HttpServlet {

   private Connection connection;
   private Statement statement;

  // set up database connection and create SQL statement
  public void init( ServletConfig config ) throws ServletException
  {
    // attempt database connection and create Statement
    try
    {

      connection=DriverManager.getConnection( "jdbc:mysql://localhost:3306/testingget","root","");

      // create Statement to query database
      statement = connection.createStatement();
    } // end try
    // for any exception throw an UnavailableException to
    // indicate that the servlet is not currently available
    catch ( Exception exception )
    {
      exception.printStackTrace();
```

```
      throw new UnavailableException( exception.getMessage() );
    } // end catch
  }  // end method init

  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
     response.setContentType("text/html;charset=UTF-8");
     PrintWriter out = response.getWriter();
     try {
     String firstName = request.getParameter("firstName").toString();
     String surname = request.getParameter("surname").toString();
     try {
     statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);

     ResultSet uprs = statement.executeQuery(
        "SELECT * FROM names");

     uprs.moveToInsertRow();
     uprs.updateString("firstname",firstName);
     uprs.updateString("lastname",surname);
     uprs.insertRow();
     uprs.beforeFirst();
  }
     catch ( SQLException sqlException )
   {
     sqlException.printStackTrace();
   }
     try
   {

     // create Statement for querying database
     statement = connection.createStatement();

     // query database
     ResultSet resultSet = statement.executeQuery(
        "SELECT * from names" );
     out.println("<html>");
     out.println("<head>");
     out.println("</head>");
     out.println("<body>");
     out.println("<p>Welcome " + firstName + " " + surname + "</p>");
     out.println( "<p>People currently in the database:</p>" );
     // process query results
     ResultSetMetaData metaData = resultSet.getMetaData();
     int numberOfColumns = metaData.getColumnCount();
     for ( int i = 1; i <= numberOfColumns; i++ )
```

```
          out.println("<label style='color:red'>"+ metaData.getColumnName( i )+"</label>" );
       out.println("</br>");
        while ( resultSet.next() )
        {
          for ( int i = 1; i <= numberOfColumns; i++ )
            out.println("<label style='color:blue'>"+ resultSet.getObject( i )+"</label>" );
          out.println("</br>");
        } // end while
       out.println("</body>");
       out.println("</html>");
     }  // end try
     catch ( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch

    }//end try

     finally {
          out.close();
       }
   }
 // close SQL statements and database when servlet terminates
  public void destroy()
  {
     // attempt to close statements and database connection
     try
     {
       statement.close();
       connection.close();
     } // end try
     // handle database exceptions by returning error to client
     catch( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch
  } // end method destroy

}
```

Handling HTTP POST Requests

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a Web page is submitted.

```
//index.jsp
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Testing POST</title>
```

```
  </head>
  <body>
    <form action="testingpost" method="post">
      <label style="color: red;"> <b>Testing Post:</b></label><br/></br>
      First Name: <input type="text" name="firstName" size="20"><br /><br/>
    <input type="submit" value="Submit"><br/><br/>
  </form>
</body>
</html>

//TestingPost
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class TestingPost extends HttpServlet {
  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      String firstName = request.getParameter("firstName").toString();
    out.println("<html>");
    out.println("<head>");
    out.println("</head>");
    out.println("<body>");
    out.println("<label style='color:red'>Welcome </label>");
    out.print("<label style='color:green'>"+firstName+"</label>");
    out.println("</body>");
    out.println("</html>");}
    finally {
      out.close();
    }
  }
}
```

Using Cookies

Now, let's develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a Web page is submitted. The example contains three files as summarized here:

File	Description
index.jsp	Allows a user to specify a value for the cookie named MyCookie.
AddCookie.java	Processes the submission of AddCookie.htm.
GetCookie.java	Displays cookie values.

```
//index.jsp
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Testing Cookies</title>
  </head>
  <body>
    </form>
    <form action="addCookie" method="post">
      <label style="color: red;"> <b>Testing Cookies<b></label><br/></br>
            <b>Enter the value for cookie</b></br>
      First Name: <input type="text" name="firstName" size="20"><br /><br/>
       Last Name:  <input type="text" name="surname" size="20"><br/><br/>
    <input type="submit" value="Submit"><br/><br/>
  </form>
    <label><b>Click below to get Cookies Value</b></label></br>
    <a href="getCookie">click here</a> <br/><br/>
</body>
</html>

//AddCookie.java
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      // Get parameter from HTTP request.
      String data = request.getParameter("firstName");
      String data1 = request.getParameter("surname");

      // Create cookie.
      Cookie cookie = new Cookie("FirstCookie", data);
      Cookie cookie1 = new Cookie("SecondCookie", data1);

      // Add cookie to HTTP response.
      response.addCookie(cookie);
      response.addCookie(cookie1);
      // Write output to browser.
      out.println("<html>");
```

```
            out.println("<head>");
            out.println("<title>Servlet AddCookie</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<B>MyCookie has been set to");
            out.println(data);
             out.println("<br/>");
             out.println(data1);
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
}

//GetCookie.java
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            Cookie[] cookies = request.getCookies();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet GetCookie</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<B>");
            for(int i = 0; i < cookies.length; i++) {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            out.println("name = " + name +
            "; value = " + value);
            out.println("</br>");
            out.println("</body>");
            out.println("</html>");
        }
        }
        finally {
```

```
        out.close();
      }
    }
}
```

Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the **getSession( )** method of HttpServletRequest. An HttpSession object is returned. This object can store a set of bindings that associate names with objects. The **setAttribute( ), getAttribute( ), getAttributeNames( ), and removeAttribute( ) methods** of HttpSession manage these bindings. It is important to note that session state is shared among all the servlets that are associated with a particular client.

```
//index.jsp
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Testing Cookies</title>
  </head>
  <body>
    <label style="color: blue"><b>Testing Session</b></label></br>
    <label><b>Click below to get Session Value</b></label></br>
    <a href="getSession">click here</a>
</body>
</html>
```

```
//GetSession.java
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

  protected void processRequest(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
      // Get the HttpSession object.
      HttpSession hs = request.getSession(true);
      // Get writer.
      // response.setContentType("text/html");
```

```
        //PrintWriter pw = response.getWriter();
        out.print("<B>");
        // Display date/time of last access.
        Date date = (Date)hs.getAttribute("date");

        // Display current date/time.

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet GetSession</title>");
        out.println("</head>");
        out.println("<body>");
       if(date != null) {
        out.print("Last access: " + date + "<br>");
        }
       date = new Date();
        hs.setAttribute("date", date);
        out.println("Current date: " + date);
        out.println("</body>");
        out.println("</html>");
     } finally {
        out.close();
     }
  }
}
```

JavaServer Pages (JSP)

In the previous chapter, you learned how to generate dynamic Web pages with servlets. You probably have already noticed in our examples that most of the code in our servlets generated output that consisted of the HTML elements that composed the response to the client. Only a small portion of the code dealt with the business logic. Generating responses from servlets requires that Web application developers be familiar with Java. However, many people involved in Web application development, such as Web site designers, do not know Java. It is difficult for people who are not Java programmers to implement, maintain and extend a Web application that consists of primarily of servlets. The solution to this problem is JavaServer Pages (JSP)an extension of servlet technology that separates the presentation from the business logic. This lets Java programmers and Web-site designers focus on their strengthswriting Java code and designing Web pages, respectively.

JavaServer Pages simplify the delivery of dynamic Web content. They enable Web application programmers to create dynamic content by reusing predefined components and by interacting with components using server-side scripting. Custom-tag libraries are a powerful feature of JSP that allows Java developers to hide complex code for database access and other useful services for dynamic Web pages in custom tags. Web sites use these custom tags like any other Web page element to take advantage of the more complex functionality hidden by the tag. Thus, Web-page designers who are not familiar with Java can enhance Web pages with powerful dynamic content and processing capabilities.

The classes and interfaces that are specific to JavaServer Pages programming are located in packages

javax.servlet.jsp and **javax.servlet.jsp.tagext**.

JavaServer Pages Overview

There are **four key components** to JSPs-**directives, actions, scripting elements and tag libraries**. **Directives** are messages to the JSP container-the server component that executes JSPs-that enable the programmer to specify page settings, to include content from other resources and to specify custom tag libraries for use in a JSP. **Actions** encapsulate functionality in predefined tags that programmers can embed in a JSP. Actions often are performed based on the information sent to the server as part of a particular client request. They also can create Java objects for use in JSP scriptlets. **Scripting elements** enable programmers to insert Java code that interacts with components in a JSP (and possibly other Web application components) to perform request processing. **Scriptlets**, one kind of scripting element, contain code fragments that describe the action to be performed in response to a user request**. Tag libraries** are part of the tag extension mechanism that enables programmers to create custom tags. Such tags enable Web page designers to manipulate JSP content without prior Java knowledge.

In some ways, JavaServer Pages look like standard XHTML or XML documents. In fact, JSPs normally include XHTML or XML markup. Such markup is known as **fixed-template data or fixed-template text.** Fixed-template data often helps a programmer decide whether to use a servlet or a JSP. Programmers tend to use JSPs when most of the content sent to the client is fixed-template data and little or none of the content is generated dynamically with Java code. Programmers typically use servlets when only a small portion of the content sent to the client is fixed-template data. In fact, some servlets do not produce content. Rather, they perform a task on behalf of the client, then invoke other servlets or JSPs to provide a response. Note that in most cases servlet and JSP technologies are interchangeable. As with servlets, JSPs normally execute as part of a Web server.

When a JSP-enabled server receives the first request for a JSP, the JSP container translates the JSP into a Java servlet that handles the current request and future requests to the JSP. Literal text in a JSP becomes string literals in the servlet that represents the translated JSP. Any errors that occur in compiling the new servlet result in translation-time errors. The JSP container places the Java statements that implement the JSP's response in method _jspService at translation time. If the new servlet compiles properly, the JSP container invokes method _jspService to process the request. The JSP may respond directly or may invoke other Web application components to assist in processing the request. Any errors that occur during request processing are known as request-time errors.

Overall, the request-response mechanism and the JSP life cycle are the same as those of a servlet. JSPs can override methods **jspInit** and **jspDestroy** (similar to servlet methods init and destroy), which the JSP container invokes when initializing and terminating a JSP, respectively. JSP programmers can define these methods using JSP declarations--part of the JSP scripting mechanism.

A Simple JSP Example

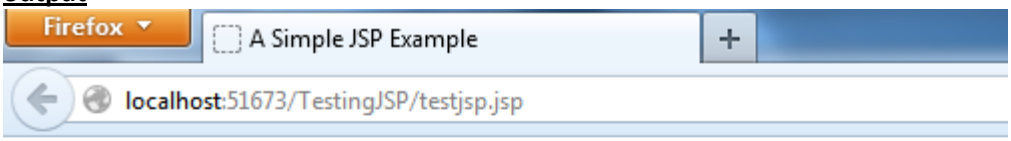
JSP expression inserting the date and time into a Web page.

//test.jsp

```
<html>
  <head>
    <meta http-equiv = "refresh" content = "60" />
     <title>A Simple JSP Example</title>
     <style type = "text/css">
       .big { font-family: helvetica, arial, sans-serif;
```

```
            font-weight: bold;
            font-size: 2em; }
    </style>
  </head>
  <body>
    <p class = "big">Simple JSP Example</p>
    <table style = "border: 6px outset;">
      <tr>
        <td style = "background-color: black;">
          <p class = "big" style = "color: cyan;">
            <!-- JSP expression to insert date/time -->
            <%= new java.util.Date() %>
          </p>
        </td>
      </tr>
    </table>
  </body
</html>
```

output



Simple JSP Example

As you can see, most of test.jsp consists of XHTML markup.In cases like this, JSPs are easier to implement than servlets. In a servlet that performs the same task as this JSP, each line of XHTML markup typically is a separate Java statement that outputs the string representing the markup as part of the response to the client. Writing code to output markup can often lead to errors.That's whhy in such scenarios JSP is preferred than Servlets.The key line in the above program is the expression

<%= new java.util.Date() %>

JSP expressions are delimited by <%= and %>. The preceding expression creates a new instance of class Date (package java.util). By default, a Date object is initialized with the current date and time. When the client requests this JSP, the preceding expression inserts the String representation of the date and time

in the response to the client. [Note: **Because the client of a JSP could be anywhere in the world, the JSP should return the date in the client locale's format**. However, the JSP executes on the server, so the server's locale determines the String representation of the Date.

We use the XHTML meta element in line 9 to set a refresh interval of 60 seconds for the document. This causes the browser to request test.jsp every 60 seconds. For each request to test.jsp, the JSP container reevaluates the expression in line 24, creating a new Date object with the server's current date and time.

When you first invoke the JSP, you may notice a brief delay as GlassFish Server translates the JSP into a servlet and invokes the servlet to respond to your request

Implicit Objects

Implicit objects provide access to many servlet capabilities in the context of a JavaServer Page. Implicit objects have **four scopes: application, page, request and session**. The JSP container owns objects with application scope. Any JSP can manipulate such objects. Objects with page scope exist only in the page that defines them. Each page has its own instances of the page-scope implicit objects. Objects with request scope exist for the duration of the request. For example, a JSP can partially process a request, then forward it to a servlet or another JSP for further processing. Request-scope objects go out of scope when request processing completes with a response to the client. Objects with session scope exist for the client's entire browsing session. Figure below describes the JSP implicit objects and their scopes.

Implicit object	Description
Application Scope	
`application`	A `javax.servlet.ServletContext` object that represents the container in which the JSP executes.
Page Scope	
`config`	A `javax.servlet.ServletConfig` object that represents the JSP configuration options. As with servlets, configuration options can be specified in a Web application descriptor.
`exception`	A `java.lang.Throwable` object that represents an exception that is passed to a JSP error page. This object is available only in a JSP error page.
`out`	A `javax.servlet.jsp.JspWriter` object that writes text as part of the response to a request. This object is used implicitly with JSP expressions and actions that insert string content in a response.
`page`	An `Object` that represents the `this` reference for the current JSP instance.
`pageContext`	A `javax.servlet.jsp.PageContext` object that provides JSP programmers with access to the implicit objects discussed in this table.
`response`	An object that represents the response to the client and is normally an instance of a class that implements `HttpServletResponse` (package `javax.servlet.http`). If a protocol other than HTTP is used, this object is an instance of a class that implements `javax.servlet.ServletResponse`.
Request Scope	
`request`	An object that represents the client request and is normally an instance of a class that implements `HttpServletRequest` (package `javax.servlet.http`). If a protocol other than HTTP is used, this object is an instance of a subclass of `javax.servlet.Servlet-Request`.
Session Scope	
`session`	A `javax.servlet.http.HttpSession` object that represents the client session information if such a session has been created. This object is available only in pages that participate in a session.

fig. JSP implicit objects.

Scripting

JavaServer Pages often present dynamically generated content as part of an XHTML document that is

sent to the client in response to a request. In some cases, the content is static but is output only if certain conditions are met during a request (e.g., providing values in a form that submits a request). **JSP programmers can insert Java code and logic in a JSP using scripting.**

Scripting Components
The JSP scripting components include **scriptlets, comments, expressions, declarations and escape sequences.**

Scriptlets are blocks of code delimited by **<% and %>.** They contain Java statements that the container places in method _jspService at translation time.

JSPs support three **comment** styles: **JSP comments, XHTML comments and scripting-language comments. JSP comments** are delimited by **<%-- and --%>.** These can be placed throughout a JSP, but not inside scriptlets. **XHTML comments** are delimited with **<!-- and -->.** These, too, can be placed throughout a JSP, but not inside scriptlets. **Scripting language comments** are currently **Java comments**, because Java currently is the only JSP scripting language. Scriptlets can use Java's **end-of-line //** comments and traditional comments (delimited by **/* and */).** JSP comments and scripting-language comments are ignored and do not appear in the response to a client. When clients view the source code of a JSP response, they will see only the XHTML comments in the source code. The different comment styles are useful for separating comments that the user should be able to see from those that document logic processed on the server.

JSP expressions are delimited by **<%= and %>** and contain a Java expression that is evaluated when a client requests the JSP containing the expression. The container converts the result of a JSP expression to a String object, then outputs the String as part of the response to the client.

Declarations, delimited by **<%! and %>,** enable a JSP programmer to define variables and methods for use in a JSP. Variables become instance variables of the servlet class that represents the translated JSP. Similarly, methods become members of the class that represents the translated JSP. Declarations of variables and methods in a JSP use Java syntax. Thus, a variable declaration must end with a semicolon, as in

```
<%! int counter = 0; %>
```

Special characters or character sequences that the JSP container normally uses to delimit JSP code can be included in a JSP as literal characters in scripting elements, fixed template data and attribute values using **escape sequences**. Figure below shows the literal character or characters and the corresponding escape sequences and discusses where to use the escape sequences.

Literal	Escape sequence	Description
<%	<\%	The character sequence <% normally indicates the beginning of a scriptlet. The <\% escape sequence places the literal characters <% in the response to the client.
%>	%\>	The character sequence %> normally indicates the end of a scriptlet. The %\> escape sequence places the literal characters %> in the response to the client.
' " \	\' \" \\	As with string literals in a Java program, the escape sequences for characters ',", and \ allow these characters to appear in attribute values. Remember that the literal text in a JSP becomes string literals in the servlet that represents the translated JSP.

fig. JSP escape sequences

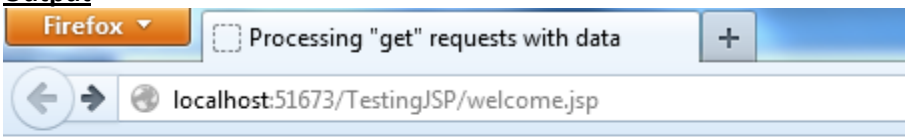
Scripting Example

//welcome.jsp

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Processing "get" requests with data</title>
  </head>
  <!-- body section of document -->
   <body>
    <% // begin scriptlet
     String name = request.getParameter( "firstName" );

      if ( name != null )
      {
    %> <%-- end scriptlet to insert fixed template data --%>

        <h1>
          Hello <%= name %>, <br />
          Welcome to JavaServer Pages!
        </h1>

    <% // continue scriptlet

      } // end if
      else {

    %> <%-- end scriptlet to insert fixed template data --%>
```

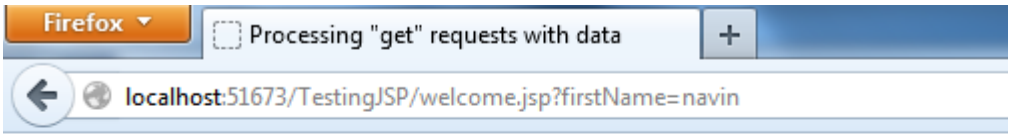
```
        <form action = "welcome.jsp" method = "get">
          <p>Type your first name and press Submit</p>

          <p><input type = "text" name = "firstName" />
            <input type = "submit" value = "Submit" />
          </p>
        </form>

    <% // continue scriptlet

      } // end else

    %> <%-- end scriptlet --%>
  </body>
</html>
```

Output



Type your first name and press Submit

navin Submit



Hello navin,
Welcome to JavaServer Pages!

Standard Actions

Standard actions provide JSP implementors with access to several of the most common tasks performed in a JSP, **such as including content from other resources, forwarding requests to other resources and interacting with JavaBean software components**. JSP containers process actions at request time. Actions are delimited by **<jsp:action> and </jsp:action>,** where **action is the standard action name**. In cases where nothing appears between the starting and ending tags, the XML empty element syntax <jsp: action /> can be used. Figure below summarizes the JSP standard actions.

Action	Description
`<jsp:include>`	Dynamically includes another resource in a JSP. As the JSP executes, the referenced resource is included and processed.
`<jsp:forward>`	Forwards request processing to another JSP, servlet or static page. This action terminates the current JSP's execution.
`<jsp:plugin>`	Allows a plug-in component to be added to a page in the form of a browser-specific `object` or `embed` HTML element. In the case of a Java applet, this action enables the browser to download and install the Java Plug-in, if it is not already installed on the client computer.
`<jsp:param>`	Used with the `include`, `forward` and `plugin` actions to specify additional name-value pairs of information for use by these actions.
JavaBean Manipulation	
`<jsp:useBean>`	Specifies that the JSP uses a JavaBean instance (i.e., an object of the class that declares the JavaBean). This action specifies the scope of the object and assigns it an ID (i.e., a variable name) that scripting components can use to manipulate the bean.
`<jsp:setProperty>`	Sets a property in the specified JavaBean instance. A special feature of this action is automatic matching of request parameters to bean properties of the same name.
`<jsp:getProperty>`	Gets a property in the specified JavaBean instance and converts the result to a string for output in the response.

fig. JSP standard actions

<jsp:include> Action

JavaServer Pages support two include mechanisms-the **<jsp:include> action** and the **include directive**. Action <jsp:include> enables dynamic content to be included in a JavaServer Page at request time. If the included resource changes between requests, the next request to the JSP containing the <jsp:include> action includes the resource's new content. On the other hand, the include directive copies the content into the JSP once, at JSP translation time. If the included resource changes, the new content will not be reflected in the JSP that used the include directive, unless that JSP is recompiled, which normally would occur only if a new version of the JSP is installed. Figure below describes the attributes of action <jsp:include>.

Attribute	Description
page	Specifies the relative URI path of the resource to include. The resource must be part of the same Web application.
flush	Specifies whether the implicit object out should be flushed before the include is performed. If true, the JspWriter out is flushed prior to the inclusion, hence you could no longer forward to another page later on. The default value is false.

fig. Action <jsp:include> attributes.

//index.jsp

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>LN TECH PVT. LTD</title>

   <style type = "text/css">
     body
     {
       font-family: tahoma, helvetica, arial, sans-serif;
     }

     table, tr, td
     {
       font-size: .9em;
       border: 3px groove;
       padding: 5px;
       background-color: yellowgreen;
     }
   </style>

  </head>
  <body>
    <table style="width: 1280px; height: 675px">
     <tr>
       <td style = "width: 215px; text-align: center">
         <img src = "LN_Tech_logo.jpg"
           width = "140" height = "93"
           alt = "LN Tech Logo" />
       </td>
       <td>
         <%-- include banner.html in this JSP --%>
         <jsp:include page = "banner.html"
```

```
            flush = "true" />
        </td>
      </tr>
      <tr>
        <td style = "width: 215px">
          <%-- include toc.html in this JSP --%>
          <jsp:include page = "toc.html" flush = "true" />
        </td>
        <td style = "vertical-align: top">
          <%-- include clock.jsp in this JSP --%>
          <jsp:include page = "clock.jsp"
            flush = "true" />
        </td>
      </tr>
    </table>
  </body>
</html>
```

//banner.html

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
<div style = "width: 580px">
  <p><b>
    LN Tech....a dedicated team of Engineers <br /> Working
    in the field of Web<br />
    welcomes you to explore our site</b>
  </p>
  <p>
    <a href = "mailto:admin@lntech.com">admin@lntech.com</a>
    <br />Baneshwor<br />Kathmandu, Nepal
  </p>
</div>
  </body>
</html>
```

//toc.html

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
```

```
<p><a href = "signup.jsp">
      <b>Sign up</b>
</a></p>
<p><a href = "http://theastutetech.com/index.php?page=about-us">
      <b>About us</b>
</a></p>

<p><a href = "http://theastutetech.com/index.php?page=services">
      <b>Services</b>
</a></p>

<p><a href = "http://theastutetech.com/index.php?page=our-works">
      <b>Our works/Porfolios</b>
</a></p>

<p><a href = "http://theastutetech.com/index.php?page=jobs">
      <b>Jobs</b>
</a></p>
<p><a href = "http://theastutetech.com/">
      <b>Home Page</b>
</a></p>

<p>Send questions or comments about this site to
  <a href = "mailto:lntech.com">
    admin@lntech.com
  </a><br />
  Copyright 2009-2012 by LN Tech Pvt Ltd.
  All Rights Reserved.
</p>
   </body>
</html>
```

//clock.jsp

```
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>Clock Page</title>
   </head>
   <body>
     <table>
  <tr>
    <td style = "background-color: blanchedalmond;">
      <p class = "big" style = "color: black; font-size: 3em;
        font-weight: bold;">

        <%-- script to determine client local and --%>
        <%-- format date accordingly           --%>
```

```
        <%
          // get client locale
          java.util.Locale locale = request.getLocale();

          // get DateFormat for client's Locale
          java.text.DateFormat dateFormat =
            java.text.DateFormat.getDateTimeInstance(
              java.text.DateFormat.LONG,
              java.text.DateFormat.LONG, locale );

        %>  <%-- end script --%>

        <%-- output date --%>
        <%= dateFormat.format( new java.util.Date() ) %>
      </p>
    </td>
  </tr>
</table>
  </body>
</html>
```

//signup.jsp

```
<!DOCTYPE html>
<html>
  <!-- head section of document -->
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Sign up Page</title>
  </head>
  <!-- body section of document -->
    <body>
   <% // begin scriptlet

      String name = request.getParameter( "firstName" );

      if ( name != null )
      {
    %> <%-- end scriptlet to insert fixed template data --%>

        <h1>
          Hello <%= name %>, <br />
          Welcome to LN Tech!
        </h1>

    <% // continue scriptlet

      } // end if
      else {
```

```
    %> <%-- end scriptlet to insert fixed template data --%>

        <form action = "signup.jsp" method = "get">
          <p>Type your first name and press Submit</p>

          <p><input type = "text" name = "firstName" />
            <input type = "submit" value = "Submit" />
          </p>
        </form>

    <% // continue scriptlet

      } // end else

    %> <%-- end scriptlet --%>
  </body>
   </body>
</html> <!-- end XHTML document -->
```

output

Remote Method Invocation(RMI)

The **Remote Method Invocation (RMI)** model represents a distributed object application. RMI allows an object inside a JVM (a client) to invoke a method on an object running on a remote JVM (a server) and have the results returned to the client.

- Therefore, RMI implies a client and a server.

The server application typically creates an object and makes it accessible remotely.

- Therefore, the object is referred to as a remote object.
- The server registers the object that is available to clients.

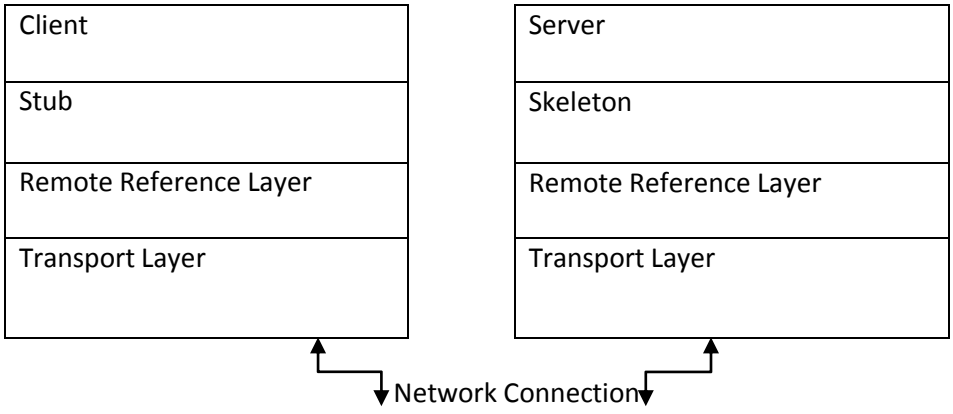
One of the ways this can be accomplished is through a naming facility provided as part of the JDK, which is called the **rmiregistry**. The server uses the registry to bind an arbitrary name to a remote object. A client application receives a reference to the object on the server and then invokes methods on it. The client looks up the name in the registry and obtains a reference to an object that is able to interface with the remote object. The reference is referred to as a remote object reference. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

RMI Architecture

The interface that the client and server objects use to interact with each other is provided through **stubs/skeleton, remote reference, and transport layers**. Stubs and skeletons are Java objects that act as proxies to the client and server, respectively.

All the network-related code is placed in the stub and skeleton, so that the client and server will not have to deal with the network and sockets in their code. The remote reference layer handles the creation of and management of remote objects. The transport layer is the protocol that sends remote object requests across the network.

A simple diagram showing the above relationships is shown below.



Developing a distributed application using RMI involves the following steps:

1. Define a remote interface
2. Implement the remote interface
3. Develop the server
4. Develop a client
5. Generate Stubs and Skeletons, start the RMI registry, server, and client

The Remote Interface

The server's job is to accept requests from a client, perform some service, and then send the results back to the client.The server must specify **an interface that defines the methods available to clients as a service**. This remote interface defines the client view of the remote object.The remote interface is always written to **extend the java.rmi.Remote interface**. Remote is a "marker" interface that identifies interfaces whose methods may be invoked from a non-local virtual machine.

```
//RemoteInterface.java
import java.rmi.*;
public interface RemoteInterface extends Remote
{
  public int add(int x,int y)throws RemoteException;
}
```

In the example above, add(int x,int y) is a remote method of the **remote interface** RemoteInterface. All methods defined in the remote interface are required to state that they **throw a RemoteException**. A RemoteException represents communication-related exceptions that may occur during the execution of a remote method call.

The Remote Object

An implementation of the RemoteInterface interface is shown below.

```
//ServerImplements.java
import java.rmi.*;
import java.rmi.server.*;
import java.lang.String;
public class ServerImplements extends UnicastRemoteObject implements RemoteInterface
 {
 public ServerImplements()throws RemoteException
 {
 super();
 }
 public int add(int x,int y)
 {
 return (x+y);
 }
}
```

The implementation is referred to as the **remote object**. The implementation class extends **UnicastRemoteObject** to link into the RMI system. This is not a requirement. A class that does not extend UnicastRemoteObject may use its exportObject() method to be linked into RMI. When a class extends UnicastRemoteObject, it **must provide a constructor declaring that it may throw a RemoteException object.** When this constructor calls super(), it activates code in UnicastRemoteObject, which performs the RMI linking and remote object initialization.

Writing the Server

```
//AdditionServer.java
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
```

```
public class  Server
{
 public static void main(String args[])
 {
 try
 {
 ServerImplements s=new ServerImplements();
 LocateRegistry.createRegistry(1099);
 Naming.rebind("SERVICE",s);
 System.out.println("Server Started ");
 }
 catch(Exception e)
 {
 System.out.println(e.getMessage());
 }
 }
}
```

The server creates the **remote object, registers it under some arbitrary name, then waits for remote requests.** The **java.rmi.registry.LocateRegistry** class allows the RMI registry service (provided as part of the JVM) to be started within the code by calling its createRegistry method.
This could have also been achieved by typing the following at a command prompt: **start rmiregistry.** The default port for RMI is 1099. The java.rmi.registry.Registry class provides two
methods for binding objects to the registry.
Naming.bind("ArbitraryName", remoteObj); throws an Exception if an object is already bound under the "ArbitrayName. "
Naming.rebind ("ArbitraryName", remoteObj); binds the object under the "ArbitraryName" if it does not exist or overwrites the object that is bound.
The example above acts as a server that creates a ServerImplements object and makes it available to clients by binding it under a name of **"SERVICE ".**

NOTE: If both the client and the server are running Java SE 5 or higher, no additional work is needed on the server side. Simply compile the **RemoteInterface.java**, **ServerImplements.java**, and **AdditionServer.java**, and the server can then be started. The reason for this is the introduction in Java SE 5 of dynamic generation of stub classes. Java SE 5 adds support for the dynamic generation of stub classes at runtime, eliminating the need to use the RMI stub compiler, **rmic**, to pre-generate stub classes for remote objects.
• Note that rmic must still be used to pre-generate stub classes for remote objects that need to support clients **running on earlier versions.**

Writing the Client

```
//Client.java
import java.rmi.*;
import java.io.*;
public class Client
{
public static void main(String args[])
 {
```

```
try
 {
String ip="rmi://127.0.0.1/SERVICE";
RemoteInterface s=
  (RemoteInterface)Naming.lookup(ip);
System.out.println("sum: "+ s.add(1,3));
 }
catch(Exception e)
 {
 System.out.println(e.getMessage());
 e.printStackTrace();
 }
 }
}
```

RMI pros and cons

Remote method invocation has significant features that CORBA doesn't possess - most notably the ability to send new objects (code and data) across a network, and for foreign virtual machines to seamlessly handle the new object. Remote method invocation has been available since JDK 1.02, and so many developers are familiar with the way this technology works, and organizations may already have systems using RMI. **Its chief limitation, however, is that it is limited to Java Virtual Machines, and cannot interface with other languages.**

Pros	cons
Portable across many platforms	Tied only to platforms with Java support
Can introduce new code to foreign JVMs	Security threats with remote code execution, and limitations on functionality enforced by security restrictions.
Java developers may already have experience with RMI (available since JDK1.02)	Learning curve for developers that have no RMI experience is comparable with CORBA
Existing systems may already use RMI - the cost and time to convert to a new technology may be prohibitive	Can only operate with Java systems - no support for legacy systems written in C++, Ada, Fortran, Cobol, and others (including future languages).

Common Object Request Broker Architecture(CORBA)

CORBA, or Common Object Request Broker Architecture, is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.

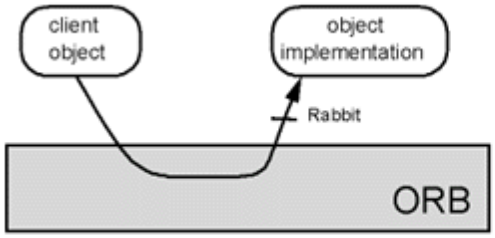
The OMG

The Object Management Group (OMG) is responsible for defining CORBA. The OMG comprises over 700 companies and organizations, including almost all the major vendors and developers of distributed object technology, including platform, database, and application vendors as well as software tool and corporate developers.

CORBA Architecture

CORBA defines an architecture for distributed objects. The basic CORBA paradigm is that of a request for services of a distributed object. Everything else defined by the OMG is in terms of this basic paradigm.

The services that an object provides are given by its interface. Interfaces are defined in OMG's Interface Definition Language (IDL). Distributed objects are identified by object references, which are typed by IDL interfaces.
The figure below graphically depicts a request. A client holds an object reference to a distributed object. The object reference is typed by an interface. In the figure below the object reference is typed by the Rabbit interface. The Object Request Broker, or ORB, delivers the request to the object and returns any results to the client. In the figure, a jump request returns an object reference typed by the AnotherObject interface.




```
interface Rabbit {
        :
    AnotherObject jump(
        in long how_high);
        :
};
```

The ORB

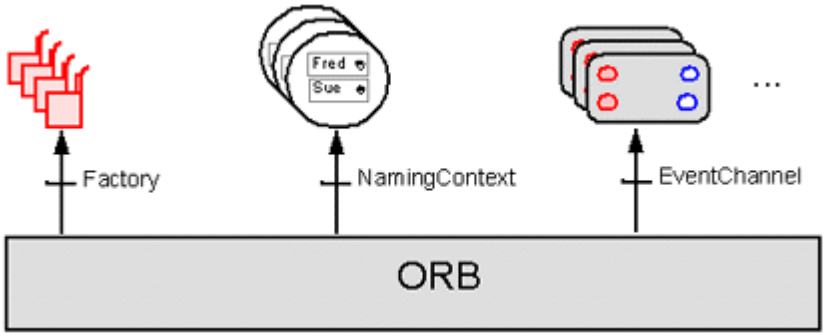
The ORB is the distributed service that implements the request to the remote object. It locates the remote object on the network, communicates the request to the object, waits for the results and when available communicates those results back to the client.
The ORB implements location transparency. Exactly the same request mechanism is used by the client and the CORBA object regardless of where the object is located. It might be in the same process with the client, down the hall or across the planet. The client cannot tell the difference.
The ORB implements programming language independence for the request. The client issuing the request can be written in a different programming language from the implementation of the CORBA object. The ORB does the necessary translation between programming languages. Language bindings are defined for all popular programming languages.

CORBA as a Standard for Distributed Objects

One of the goals of the CORBA specification is that clients and object implementations are portable. The CORBA specification defines an application programmer's interface (API) for clients of a distributed object as well as an API for the implementation of a CORBA object. This means that code written for one vendor's CORBA product could, with a minimum of effort, be rewritten to work with a different vendor's product. However, the reality of CORBA products on the market today is that CORBA clients are portable but object implementations need some rework to port from one CORBA product to another.
CORBA 2.0 added interoperability as a goal in the specification. In particular, CORBA 2.0 defines a network protocol, called **IIOP (Internet Inter-ORB Protocol),** that allows clients using a CORBA product from any vendor to communicate with objects using a CORBA product from any other vendor. IIOP works across the Internet, or more precisely, across any TCP/IP implementation.
Interoperability is more important in a distributed system than portability. IIOP is used in other systems that do not even attempt to provide the CORBA API. In particular, IIOP is used as the transport protocol for a version of Java RMI (so called "RMI over IIOP"). Since EJB is defined in terms of RMI, it too can use IIOP. Various application servers available on the market use IIOP but do not expose the entire CORBA API. Because they all use IIOP, programs written to these different API's can interoperate with each other and with programs written to the CORBA API.

CORBA Services

Another important part of the CORBA standard is the definition of a set of distributed services to support the integration and interoperation of distributed objects. As depicted in the graphic below, the services, known as CORBA Services or COS, are defined on top of the ORB. That is, they are defined as standard CORBA objects with IDL interfaces, sometimes referred to as "Object Services."



There are several CORBA services. Below is a brief description of each:

Service	Description
Object life cycle	Defines how CORBA objects are created, removed, moved, and copied
Naming	Defines how CORBA objects can have friendly symbolic names
Events	Decouples the communication between distributed objects
Relationships	Provides arbitrary typed n-ary relationships between CORBA objects
Externalization	Coordinates the transformation of CORBA objects to and from external media
Transactions	Coordinates atomic access to CORBA objects
Concurrency Control	Provides a locking service for CORBA objects in order to ensure serializable access
Property	Supports the association of name-value pairs with CORBA objects
Trader	Supports the finding of CORBA objects based on properties describing the service offered by the object
Query	Supports queries on objects

CORBA Products

CORBA is a specification; it is a guide for implementing products. Several vendors provide CORBA products for various programming languages. The CORBA products that support the Java programming language include:

ORB	Description
The Java 2 ORB	The Java 2 ORB comes with Sun's Java 2 SDK. It is missing several features.
VisiBroker for Java	A popular Java ORB from Inprise Corporation. VisiBroker is also embedded in other products. For example, it is the ORB that is embedded in the Netscape Communicator browser.
OrbixWeb	A popular Java ORB from Iona Technologies.
WebSphere	A popular application server with an ORB from IBM.
Netscape Communicator	Netscape browsers have a version of VisiBroker embedded in them. Applets can issue request on CORBA objects without downloading ORB classes into the browser. They are already there.
Various free or shareware ORBs	CORBA implementations for various languages are available for download on the web from various sources.

CORBA pros and cons

CORBA is gaining strong support from developers, because of its ease of use, functionality, and portability across language and platform. CORBA is particularly important in large organizations, where many systems must interact with each other, and legacy systems can't yet be retired. CORBA provides the connection between one language and platform and another - **its only limitation is that a language must have a CORBA implementation written for it.** CORBA also appears to have a performance increase over RMI, which makes it an attractive option for systems that are accessed by users who require real-time interaction.

Pros	Cons
Services can be written in many different languages, executed on many different platforms, and accessed by any language with an interface definition language (IDL) mapping	Describing services require the use of an interface definition language (IDL) which must be learned. Implementing or using services require an IDL mapping to your required language - writing one for a language that isn't supported would take a large amount of work.
With IDL, the interface is clearly separated from implementation, and developers can create different implementations based on the same interface.	IDL to language mapping tools create code stubs based on the interface - some tools may not integrate new changes with existing code.
CORBA supports primitive data types, and a wide range of data structures, as parameters	CORBA does not support the transfer of objects, or code.
CORBA is ideally suited to use with legacy systems, and to ensure that applications written now will be accessible in the future.	The future is uncertain - if CORBA fails to achieve sufficient adoption by industry, then CORBA implementations become the legacy systems.
CORBA is an easy way to link objects and systems together.	Some training is still required, and CORBA specifications are still in a state of flux.