Nested Classes

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

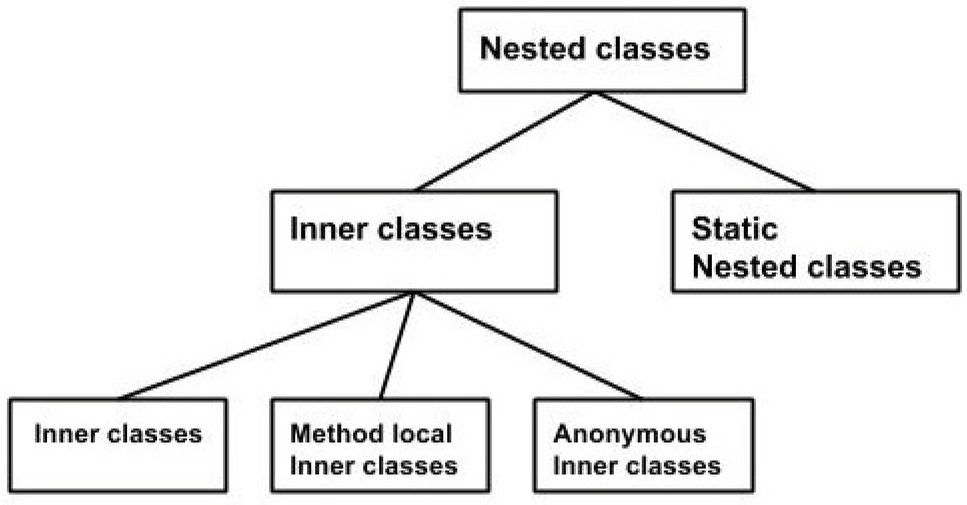
Syntax

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

```
class Outer_Demo {
  class Nested_Demo {
  }
}
```

Nested classes are divided into two types –

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



- ## Inner Classes (Non-static Nested Classes)

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

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

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

❖ *Member Inner Class*

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

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

Example

```
class Outer_Demo {
  int num;
  // inner class
  private class Inner_Demo {
    public void print() {
      System.out.println("This is an inner class");
    } }
   // Accessing he inner class from the method within
  void display_Inner() {
    Inner_Demo inner = new Inner_Demo();
    inner.print();
  }
}
 public class My_class {
  public static void main(String args[]) {
    // Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();
    // Accessing the display_Inner() method.
    outer.display_Inner();
  }
}
```

Output

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

Accessing the Private Members

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

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

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

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

Example

```
class Outer_Demo {
   // private variable of the outer class
   private int num = 175;
     // inner class
   public class Inner_Demo {
      public int getNum() {
         System.out.println("This is the getnum method of the inner class");
         return num;
      }
   }
}
public class My_class2 {
   public static void main(String args[]) {
      // Instantiating the outer class
      Outer_Demo outer = new Outer_Demo();
      // Instantiating the inner class
      Outer_Demo.Inner_Demo inner = outer.new Inner_Demo();
      System.out.println(inner.getNum());
   }
}
```

Output

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

❖ Method-local Inner Class

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method. A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

Example

```
public class Outerclass {
  // instance method of the outer class
  void my_Method() {
    int num = 23;
    // method-local inner class
    class MethodInner_Demo {
      public void print() {
        System.out.println("This is method inner class "+num);
      }
    } // end of inner class
    // Accessing the inner class
    MethodInner_Demo inner = new MethodInner_Demo();
    inner.print();
  }
  public static void main(String args[]) {
    Outerclass outer = new Outerclass();
    outer.my_Method();
  }
}
```

Output

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

❖ **Anonymous Inner Class**

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

It is a type of inner class which:

1. has no name
2. can be instantiated only once
3. is usually declared inside a method or a code block ,a curly braces ending with semicolon.
4. is accessible only at the point where it is defined.
5. does not have a constructor simply because it does not have a name

6. cannot be static

The syntax of an anonymous inner class is as follows–

Syntax

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

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

Example 1

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

Output

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

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

Example 2

```
// Java program to demonstrate accessing a inner class
 // outer class
class OuterClass
{
    // static member
    static int outer_x = 10;
    // instance(non-static) member
    int outer_y = 20;
    // private member
    private int outer_private = 30;
    // inner class
    class InnerClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);
            // can also access non-static member of outer class
            System.out.println("outer_y = " + outer_y);
            // can also access private member of outer class
            System.out.println("outer_private = " + outer_private);
         }
    }
}
public class InnerClassDemo
{
    public static void main(String[] args)
    {
        // accessing an inner class
        OuterClass outerObject = new OuterClass();
       OuterClass.InnerClass innerObject = outerObject.newInnerClass();
        innerObject.display();
        }
}
```

Output:

```
outer_x = 10
outer_y = 20
outer_private = 30
```

❖ Anonymous Inner Class as Argument

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

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

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

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

Example

// interface

```
interface Message {
   String greet();
}
public class My_class {
   // method which accepts the object of interface Message
   public void displayMessage(Message m) {
      System.out.println(m.greet() +
         ", This is an example of anonymous inner class as an argument");
   }
   public static void main(String args[]) {
      // Instantiating the class
      My_class obj = new My_class();
      // Passing an anonymous inner class as an argument
      obj.displayMessage(new Message() {
         public String greet() {
            return "Hello";
         }
      });
   }}
```

Output

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

Example:

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

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

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

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

• Static Nested Class

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

Syntax

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

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

Example 1

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

Output

```
This is my nested class
```

Example 2

```
// Java program to demonstrate accessing
// a static nested class

// outer class
class OuterClass
{
    // static member
    static int outer_x = 10;
   // instance(non-static) member
    int outer_y = 20;
         // private member
    private static int outer_private = 30;
         // static nested class
    static class StaticNestedClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);
            // can access display private static member of outer class
            System.out.println("outer_private = " + outer_private);
             // The following statement will give compilation error
            // as static nested class cannot directly access non-static membera
            // System.out.println("outer_y = " + outer_y);
         }
    }
}
public class StaticNestedClassDemo
{
    public static void main(String[] args)
    {  // accessing a static nested class
       OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
        nestedObject.display();
       }
}
```

Output:

```
outer_x = 10

outer_private = 30
```

✓ Difference between static and inner(non-static nested) classes

- Static nested classes do not directly have access to other members(non-static variables and methods) of the enclosing class because as it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

- Non-static nested classes (inner classes) has access to all members(static and non-static variables and methods, including private) of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can ***wrap a primitive value into a wrapper class object***.

Features of the Java wrapper Classes.

1. Wrapper classes convert numeric strings into numeric values.
2. The way to store primitive data in an object.
3. The valueOf() method is available in all wrapper classes except Character
4. All wrapper classes have typeValue() method. This method returns the value of the object as its primitive type.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

The following programs demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

Example 1: Converting a primitive type to Wrapper object

```
public class JavaExample{
   public static void main(String args[]){
       //Converting int primitive into Integer object
       int num=100;
       Integer obj=Integer.valueOf(num);

       System.out.println(num+ " "+ obj);
   }
}
```

Output:

```
100 100
```

Example 2: Converting Wrapper class object to Primitive

```
public class JavaExample{
   public static void main(String args[]){
       //Creating Wrapper class object
       Integer obj = new Integer(100);

       //Converting the wrapper object to primitive
       int num = obj.intValue();

       System.out.println(num+ " "+ obj);
   }
}
```

Output:

```
100 100
```

Autoboxing and Unboxing

***Autoboxing*:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing.
For example – conversion of int to Integer, long to Long, double to Double etc.
in the example 2, this line boxes the value 100 into an **Integer**:

```
Integer obj = new Integer(100);
```

Example:

```
// Java program to demonstrate Autoboxing
 import java.util.ArrayList;
class Autoboxing
{ public static void main(String[] args)
    { char ch = 'y';
      // Autoboxing- primitive to Character object conversion
        Character a = ch;
         ArrayList<Integer> arrayList = new ArrayList<Integer>();
         // Autoboxing because ArrayList stores only objects
        arrayList.add(25);
         // printing the values from object
        System.out.println(arrayList.get(0));
    }}
```

Output:

```
25
```

Unboxing: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc. In example2, the program unboxes the value in **num** with this statement:

```
int num = obj.intValue();
```

Example:

```
// Java program to demonstrate Unboxing
import java.util.ArrayList;
 class Unboxing
{ public static void main(String[] args)
    {     Character ch = 'f';
         // unboxing - Character object to primitive conversion
        char a = ch;
         ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(24);
         // unboxing because get method returns an Integer object
        int num = arrayList.get(0);
         // printing the values from primitive data types
        System.out.println(num);
    }}
```

Output:

```
24
```

Example:

```
// Java program to demonstrate Wrapping and UnWrapping
// in Java Classes
class WrappingUnwrapping
{ public static void main(String args[])
   { //  byte data type
      byte a = 1;
       // wrapping around Byte object
      Byte byteobj = new Byte(a);
       // int data type
      int b = 10;
       //wrapping around Integer object
      Integer intobj = new Integer(b);
       // float data type
      float c = 18.6f;
       // wrapping around Float object
      Float floatobj = new Float(c);
       // double data type
      double d = 250.5;
       // Wrapping around Double object
      Double doubleobj = new Double(d);
       // char data type
      char e='a';
       // wrapping around Character object
      Character charobj=e;
       //  printing the values from objects
     System.out.println("Values of Wrapper objec(printin as objects)");
      System.out.println("Byte object byteobj:  " + byteobj);
      System.out.println("Integer object intobj:  " + intobj);
      System.out.println("Float object floatobj:  " + floatobj);
      System.out.println("Double object doubleobj:  " +doubleobj);
      System.out.println("Character object charobj:  " + charobj);
      // objects to data types (retrieving data types from objects)
      // unwrapping objects to primitive data types
      byte bv = byteobj;
      int iv = intobj;
      float fv = floatobj;
      double dv = doubleobj;
      char cv = charobj;
       // printing the values from data types
      System.out.println("Unwrapped values (printing as data types)");
      System.out.println("byte value, bv: " + bv);
      System.out.println("int value, iv: " + iv);
```

```
        System.out.println("float value, fv: " + fv);
        System.out.println("double value, dv: " + dv);
        System.out.println("char value, cv: " + cv);
    }
}
```

Output:

```
Values of Wrapper objects (printing as objects)
Byte object byteobj:  1
Integer object intobj:  10
Float object floatobj:  18.6
Double object doubleobj:  250.5
Character object charobj: a
Unwrapped values (printing as data types)
byte value, bv: 1
int value, iv: 10
float value, fv: 18.6
double value, dv: 250.5
char value, cv: a
```

example:

Pass-by-reference and pass-by-value

```
public class IntegerWrapper {
  public int objectInt = 0;
}
public class Hello {
  static int primitiveInt = 0;
  static IntegerWrapper intWrapper = new IntegerWrapper();
  public static void main(String[] args) throws Exception {
    passBy(primitiveInt, intWrapper);
    System.out.println("primitiveInt = " + primitiveInt +
        "; intWrapper.objectInt = " + intWrapper.objectInt);
  }
    public static void passBy(int primitiveInt, IntegerWrapper intWrapper) {
    primitiveInt++;
    intWrapper.objectInt++;
  }
}
```

PassByTest with a return value for the passBy method

```
public class Hello {

  static int primitiveInt = 0;
  static IntegerWrapper intWrapper = new IntegerWrapper();

  public static void main(String[] args) {
    int a=passBy(primitiveInt, intWrapper);
    System.out.println("primitiveInt = " + a +
        "; intWrapper.objectInt = " + intWrapper.objectInt);
  }
    public static int passBy(int primitiveInt, IntegerWrapper intWrapper) {
    primitiveInt++;
    intWrapper.objectInt++;
    return primitiveInt;
  }
}
```

Multithreading

Before we talk about **multithreading**, let’s discuss threads. A thread is a light-weight smallest part of a process that can run concurrently with the other parts(other threads) of the same process. Threads are independent because they all have separate path of execution that’s the reason if an exception occurs in one thread, it doesn’t affect the execution of other threads. All threads of a process share the common memory. **The process of executing multiple threads simultaneously is known as multithreading**.

Let’s summarize the discussion in points:
1. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.
2. Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.
3.A thread can be in one of the following states:

NEW – A thread that has not yet started is in this state.

RUNNABLE – A thread executing in the Java virtual machine is in this state.

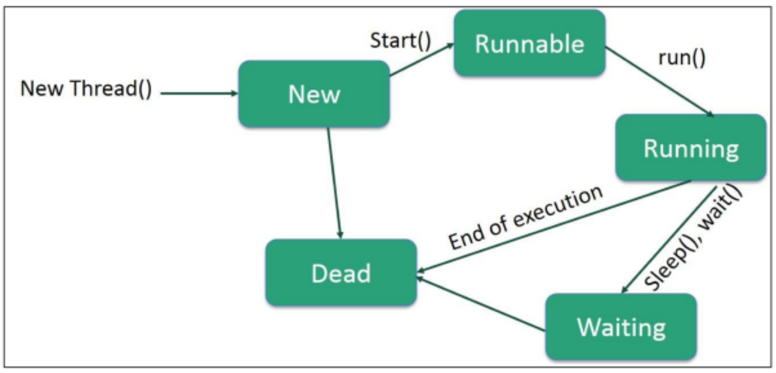
BLOCKED – A thread that is blocked waiting for a monitor lock is in this state.

WAITING – A thread that is waiting indefinitely for another thread to perform a particular action is in this state.

TIMED_WAITING – A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.

TERMINATED – A thread that has exited is in this state.

A thread can be in only one state at a given point in time.



Multitasking vs Multithreading vs Multiprocessing vs parallel processing:

If you are new to java you may get confused among these terms as they are used quite frequently when we discuss multithreading. Let's talk about them in brief.

Multitasking: Ability to execute more than one task at the same time is known as multitasking.

Multithreading: We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

Multiprocessing: It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

Parallel Processing: It refers to the utilization of multiple CPUs in a single computer system.

Creating a thread in Java

There are two ways to create a thread in Java:
1) By extending Thread class.
2) By implementing Runnable interface.
Before we begin with the programs(code) of creating threads, let's have a look at these methods of Thread class. We have used few of these methods in the example below.

- getName(): It is used for Obtaining 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

Method 1: Thread creation by extending Thread class

Example 1:

```
class MultithreadingDemo extends Thread{
  public void run(){
    System.out.println("My thread is in running state.");  }
  public static void main(String args[]){
     MultithreadingDemo obj=new MultithreadingDemo();
     obj.start();  }  }
```

Output:

```
My thread is in running state.
```

Example 2:

```
class Count extends Thread
{
   Count()
   {
     super("my extending thread");
     System.out.println("my thread created" + this);
     start();
   }
   public void run()
   {
     try
     {
        for (int i=0 ;i<10;i++)
        {
           System.out.println("Printing the count " + i);
           Thread.sleep(1000);
        }
     }
     catch(InterruptedException e)
     {
        System.out.println("my thread interrupted");
     }
     System.out.println("My thread run is over" );
   }
}
class ExtendingExample
{
   public static void main(String args[])
   {
      Count cnt = new Count();
      try
      {
         while(cnt.isAlive())
         {
           System.out.println("Main thread will be alive till the child thread
is live");
           Thread.sleep(1500);
         }
      }
      catch(InterruptedException e)
      { System.out.println("Main thread interrupted");}
```

```
    System.out.println("Main thread's run is over" );}}
```

Output:

```
my thread createdThread[my runnable thread,5,main]
Main thread will be alive till the child thread is live
Printing the count 0
Printing the count 1
Main thread will be alive till the child thread is live
Printing the count 2
Main thread will be alive till the child thread is live
Printing the count 3
Printing the count 4
Main thread will be alive till the child thread is live
Printing the count 5
Main thread will be alive till the child thread is live
Printing the count 6
Printing the count 7
Main thread will be alive till the child thread is live
Printing the count 8
Main thread will be alive till the child thread is live
Printing the count 9
mythread run is over
Main thread run is over
```

Method 2: Thread creation by implementing Runnable Interface

How to launch a new thread:

1 Make a Runnable object (the thread's job)

```
Runnable threadJob = new MyRunnable();
```

Runnable is an interface you'll learn about on the next page. You'll write a class that implements the Runnable interface, and that class is where you'll define the work that a thread will perform. In other words, the method that will be run from the thread's new call stack.

2 Make a Thread object (the worker) and give it a Runnable (the job)

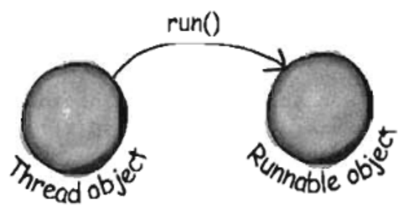
```
Thread myThread = new Thread(threadJob);
```

Pass the new Runnable object to the Thread constructor. This tells the new Thread object which method to put on the bottom of the new stack—the Runnable's run() method.

3 Start the Thread

```
myThread.start();
```

Nothing happens until you call the Thread's start() method. That's when you go from having just a Thread instance to having a new thread of execution. When the new thread starts up, it takes the Runnable object's run() method and puts it on the bottom of the new thread's stack.



A Simple Example

```
class MultithreadingDemo implements Runnable{
  public void run(){
    System.out.println("My thread is in running state.");
  }
  public static void main(String args[]){
     MultithreadingDemo obj=new MultithreadingDemo();
     Thread tobj =new Thread(obj);
     tobj.start();
 }
}
```

Output:

```
My thread is in running state.
```

Example Program 2:.

```
class Count implements Runnable
{
  Thread mythread ;
  Count()
  {
    mythread = new Thread(this, "my runnable thread");
    System.out.println("my thread created" + mythread);
    mythread.start();
  }
  public void run()
  {
    try
    {
      for (int i=0 ;i<10;i++)
      {
        System.out.println("Printing the count " + i);
        Thread.sleep(1000);
      }
    }
   catch(InterruptedException e)
   {
     System.out.println("my thread interrupted");
   }
   System.out.println("mythread run is over" );
  }}
class RunnableExample
{
   public static void main(String args[])
   {
     Count cnt = new Count();
     try
     {
       while(cnt.mythread.isAlive())
       {
         System.out.println("Main thread will be alive till the child thread
is live");
         Thread.sleep(1500);
       }
     }
     catch(InterruptedException e)
     {
       System.out.println("Main thread interrupted");
     }
     System.out.println("Main thread run is over" );
   }
}
```

Output:

```
my thread createdThread[my runnable thread,5,main]
Main thread will be alive till the child thread is live
Printing the count 0
Printing the count 1
Main thread will be alive till the child thread is live
Printing the count 2
Main thread will be alive till the child thread is live
Printing the count 3
Printing the count 4
Main thread will be alive till the child thread is live
Printing the count 5
Main thread will be alive till the child thread is live
Printing the count 6
Printing the count 7
Main thread will be alive till the child thread is live
Printing the count 8
Main thread will be alive till the child thread is live
Printing the count 9
mythread run is over
Main thread run is over
```

Thread priorities

- Thread priorities are the integers which decide how one thread should be treated with respect to the others.
- Thread priority decides when to switch from one running thread to another, process is called context switching
- A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
- To set the priority of the thread setPriority() method is used which is a method of the class Thread Class.

In place of defining the priority in integers, we can use MIN_PRIORITY, NORM_PRIORITY or MAX_PRIORITY.

Example:

```
// Demonstrate thread priorities.
class Priority implements Runnable {
  int count;
  Thread thrd;
  static boolean stop = false;
  static String currentName;
 /* Construct a new thread. Notice that this
     constructor does not actually start the
      threads running. */
  Priority(String name) {
    thrd = new Thread(this, name);
    count = 0;
    currentName = name;
  }
// Begin execution of new thread.
public void run() {
  System.out.println(thrd.getName() + " starting.");
  do {
    count++;
    if(currentName.compareTo(thrd.getName()) != 0) {
    currentName = thrd.getName();
    System.out.println("In " + currentName);
   }
  } while(stop == false && count < 10000000);
    stop = true;
    System.out.println("\n" + thrd.getName() +
    " terminating.");
  }
 }
class PriorityDemo {
  public static void main(String args[]) {
  Priority mt1 = new Priority("High Priority");
  Priority mt2 = new Priority("Low Priority");
  // set the priorities
  mt1.thrd.setPriority(Thread.NORM_PRIORITY+2);
  mt2.thrd.setPriority(Thread.NORM_PRIORITY-2);
 // start the threads
 mt1.thrd.start();
 mt2.thrd.start();
 try {
  mt1.thrd.join();
  mt2.thrd.join();
 }
 catch(InterruptedException exc) {
  System.out.println("Main thread interrupted.");
 } System.out.println("\nHigh priority thread counted to " +
  mt1.count);
  System.out.println("Low priority thread counted to " +
  mt2.count);
 }
}
```

Output:

```
High Priority starting.
In High Priority
Low Priority starting.
In Low Priority
In High Priority
High Priority terminating.
Low Priority terminating.
High priority thread counted to 10000000
Low priority thread counted to 8183
```

In this run, the high-priority thread got a vast majority of the CPU time. Of course, the exact output produced by this program will depend upon the speed of your CPU, the operating system you are using, and the number of other tasks running in the system.

Methods: isAlive() and join()

- In all the practical situations main thread should finish last else other threads which have spawned from the main thread will also finish.
- To know whether the thread has finished we can call isAlive() on the thread which returns true if the thread is not finished.
- Another way to achieve this by using join() method, this method when called from the parent thread makes parent thread wait till child thread terminates.
- These methods are defined in the Thread class.
- We have used isAlive() method in the above examples too.

Synchronization

- Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
- When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.
- To implement the synchronous behavior java has synchronous method. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread come out of the synchronized block.
- When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the synchronized to the appropriate methods. In java we have the solution for this, put the calls to the methods (which needs to be synchronized) defined by this class inside a synchronized block in following manner.

```
Synchronized(object)
{
   // statement to be synchronized
}
```

Please notice that constructors cannot be synchronized (using the synchronized keyword with a constructor raises compiler error) because only the thread which creates an instance has access to it while instance is being constructed.
We have few methods through which java threads can communicate with each other. These methods are wait(), notify(), notifyAll(). All these methods can only be called from within a synchronized method.
1) To understand synchronization java has a concept of monitor. Monitor can be thought of as a box which can hold only one thread. Once a thread enters the monitor all the other threads have to wait until that thread exits the monitor.
2) wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
3) notify() wakes up the first thread that called wait() on the same object.

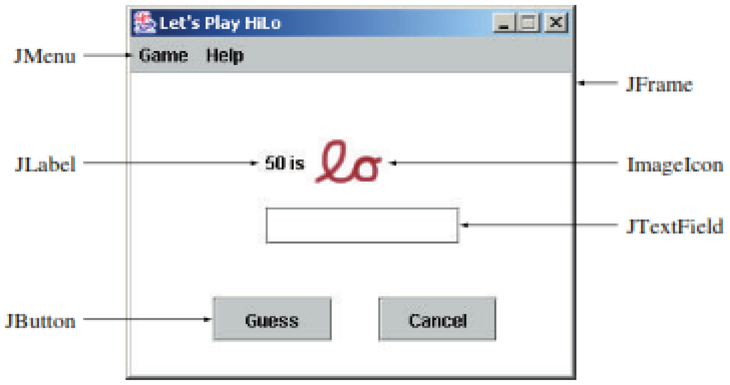
notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

Example:

```
class TestSync implements Runnable {
   private int balance;
   public void run()
   for(int i = 0; i < 50; i++) {
     increment () ;
     System.out.println("balance is " + balance);
 }}
public  synchrounaized void increment() {
int i = balance;
balance = i +1;
}}
public class TestSyncTest {
public static void main (String[] args) {
TestSync job = new TestSync();
Thread a = new Thread(job);
Thread b = new Thread(job) ;
a.start();
b.start();
}}
```

Graphical User Interface (GUI):

The type of user interface we cover in this subject is called a ***graphical user interface*** (GUI). In contrast, the user interface that uses **System.in** and **System.out** exclusively is the called the *non-GUI*, or *console user interface*. In Java, GUI-based programs are implemented by using the classes from the standard **javax.swing** and **java.awt** packages. We will refer to them collectively as *GUI classes*. When we need to differentiate them, we will refer to the classes from **javax.swing** as ***Swing classes*** and those from **java.awt** as ***AWT classes***. Some of the GUI objects from the **javax.swing** package are shown in Figure

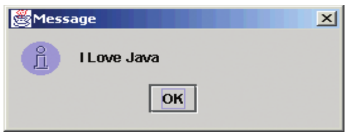


Simple GUI I/O with JOptionPane:

One of the easiest ways to provide a simple GUI-based input and output is by using the **JOptionPane** class. For example, when we execute the statement :

```
JOptionPane.showMessageDialog(null, "I Love Java");
```

the dialog appears on the center of the screen.



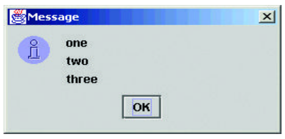
In a GUI environment, there are basically two types of windows:
1-a *general-purpose frame*
2- a *special-purpose dialog*.
In Java, we use a **JFrame** object for a frame window and a **JDialog** object for a dialog. The first argument to the **showMessageDialog** method is a frame object that controls this dialog, and the second argument is the text to display. In the example statement, we pass **null**,a reserved word, meaning there is no frame object. If we pass **null** as the first argument, the dialog appears on the center of the screen. If we pass a frame object, then the dialog is positioned at the center of the frame.

Example:

```
/*Shows a Message Dialog*/
import javax.swing.*;
class TestShowMessageDialog {
public static void main(String[] args){
JFrame jFrame;
jFrame = new JFrame();
jFrame.setSize(400,300);
jFrame.setVisible(true);
JOptionPane.showMessageDialog(jFrame, "How are you?");
JOptionPane.showMessageDialog(null, "Good Bye");
}}
```

Notice that we are not creating an instance of the **JDialog** class directly by ourselves. However, when we call the **showMessageDialog** method, the **JOptionPane** class is actually creating an instance of **JDialog** internally. Notice that **showMessageDialog** is a class method and therefore we are not creating a **JOptionPane** object.
If we need a more complex dialog, then we create an instance of **JDialog**. But for a simple display of text, calling the **showMessageDialog** class method of **JOptionPane** would sufice. If we want to display multiple lines of text, we can use a special character sequence **\n** to separate the lines, as in

```
    JOptionPane.showMessageDialog(null, "one\ntwo\nthree") ;
```



Message

one
two
three

We can also use the **JOptionPane** class for input by using its **showInputDialog** method. For example, when we execute

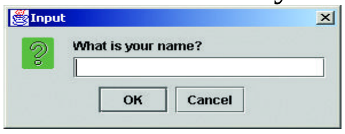
JOptionPane.showInputDialog(null, "Enter text:");

the dialog appears on the screen. To assign the name **input** to an input string, we write:

String input;
input = JOptionPane.showInputDialog(null, "Enter text:");

- *Unlike the Scanner class that supports different input methods for specific data types, that is, nextInt and nextDouble, the JOptionPane supports only a string.*

An input dialog that appears as a result of calling the showInputDialog class method of the JOptionPane class with “What is your name?” as the method’s second argument



Input

What is your name?

Customizing Frame Windows

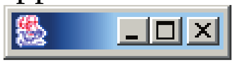
To create a customized user interface, we often define a subclass of the **JFrame** class. The helper class **MainWindow** we used in the Sample Development for example, is a subclass of the **JFrame** class. The **JFrame** class contains the most basic functionalities to support features found in any frame window, such as minimizing the window, moving the window and resizing the window.

In writing practical programs, we normally do not create an instance of the **JFrame** class because a **JFrame** object is not capable of doing anything meaningful. For example, if we want to use a frame window for a word processor, we need a frame window capable of allowing the user to enter, cut, and paste text; change font; print text; and so forth. To design such a frame window, we would define a subclass of the **JFrame** class and add methods and data members to implement the needed *functionalities.*

Before we show sample subclasses of **JFrame**, let'sfirst look at the following program which displays a default **JFrame** object on the screen:

```
/*Displays a default JFrame window*/
import javax.swing.*;
class DefaultJFrame {
public static void main(String[] args){
JFrame defaultJFrame;
defaultJFrame = new JFrame();
defaultJFrame.setVisible(true);
}}
```

When this program is executed, a default **JFrame** object appears on the screen. Since no methods (other than **setVisible**) to set the properties of the **JFrame** object (such as its title, location, and size) are called, a very small default **JFrame** object appears at the top left corner of the screen.



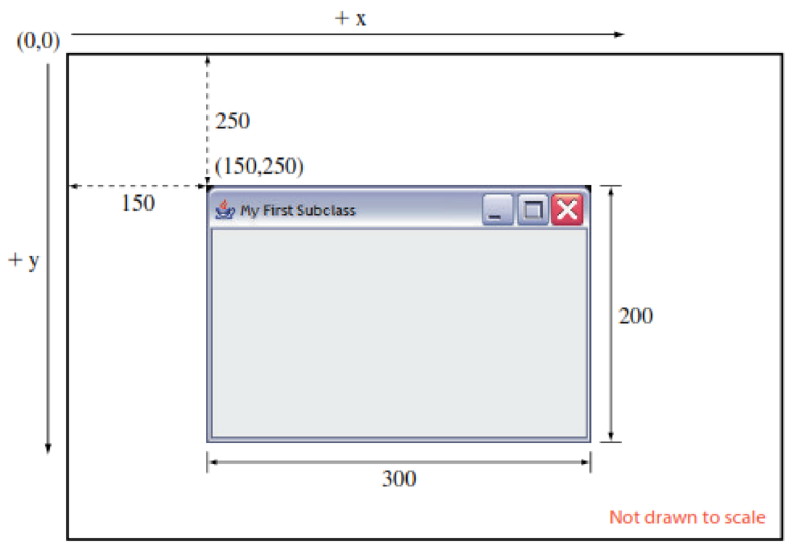
Now let's define a subclass of the **JFrame** class and add some default characteristics. To define a subclass of another class, we declare the subclass with the reserved word **extends**. So, to define a class named **JFrameSubclass1** as a subclass of **JFrame**, we declare the subclass as

```
class JFrameSubclass1 extends JFrame {
...
}
```

For the **JFrameSubclass1** class, we will add the following default characteristics:

• The title is set to **My First Subclass**.
• The program terminates when the Close box is clicked.

• The size of the frame is set to 300 pixels wide and 200 pixels high.
• The frame is positioned at screen coordinate (150, 250).
All these properties are set inside the default constructor. To set the frame's title, we pass the title to the **setTitle** method. To set the frame's size, we pass its width and height to the **setSize** method. To position the frame's top left corner to the coordinate (*x*, *y*), we pass the values *x* and *y* to the **setLocation** method. Finally, to terminate



```
import javax.swing.*;
class JFrameSubclass1 extends JFrame {
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 200;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
public JFrameSubclass1 ( ) {
//set the frame default properties
setTitle ("My First Subclass");
setSize (FRAME_WIDTH, FRAME_HEIGHT);
setLocation (FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
//register 'Exit upon closing' as a default close operation
setDefaultCloseOperation( EXIT_ON_CLOSE );}}
class TestJFrameSubclass {
public static void main(String[] args){
JFrameSubclass1 myFrame;
myFrame = new JFrameSubclass1();
myFrame.setVisible(true);
}}
```

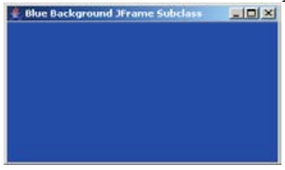
Let's define another subclass named **JFrameSubclass2** that has a blue background color instead. We will define this class as an instantiable

main class so we don't have to define a separate main class. To make the background appear in blue, we need to access the content pane of a frame. A frame's ***content pane*** designates the area of the frame that excludes the title and menu bars and the border.
It is the area we can use to display the content (text, image, etc.). We access the content pan of a frame by calling the frame's **getContentPane** method. And to change the background color to blue,we call the content pane's **setBackground** method. We carry out these operations in the private **changeBkColor** method of **JFrameSubclass2**.

Example:

```
import javax.swing.*;
import java.awt.*;
class JFrameSubclass2 extends JFrame {
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 200;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
public JFrameSubclass2() {
//set the frame default properties
setTitle    ("Blue Background JFrame Subclass");
setSize     (FRAME_WIDTH, FRAME_HEIGHT);
setLocation (FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
//register 'Exit upon closing' as a default close operation
setDefaultCloseOperation(EXIT_ON_CLOSE);
changeBkColor();
}
private void changeBkColor() {
Container c = getContentPane();
c.setBackground(Color.BLUE);
}}
public static void main(String[] args){
JFrameSubclass2 frame = new JFrameSubclass2();
frame.setVisible(true);}
```

Running the program will result in the frame appearing on the screen.

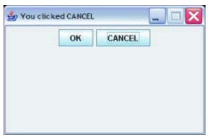


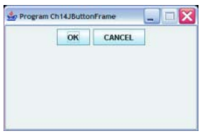
GUI Programming Basics

In this section, we will develop a sample frame window that illustrates the fundamentals of GUI programming. The sample frame window has two buttons labeled **CANCEL** and **OK**. When you click the **CANCEL** button, the window's title is changed to **You clicked CANCEL**. Likewise, when you click the **OK** button, the window's title is changed to **You clicked OK**. Figure shows the window when it isfirst opened and after the **CANCEL** button is clicked.

There are two key aspects involved in GUI programming. One is the placement of GUI objects on the content pane of a frame, and the other is the handling of events generated by these GUI objects. We will develop the sample program in two steps.

First we will define a **JFrame** subclass called **JButtonFrame** to show how the two buttons labeled **OK** and **CANCEL** are placed on the frame. Then we will implement another subclass called **JButtonEvents** to show how the button events are processed to change the frame's title. **Button Placement** The type of button we use here is called a ***pushbutton***. we will simply call them buttons. To use a button in a program,we create an instance of the **javax.swing.JButton** class. We will create two buttons and place them on the frame's content pane in the constructor. Let's name the two buttons **cancelButton** and **okButton**. We declare and create these buttons in the following manner:





```
import javax.swing.*;
JButton  cancelButton, okButton;
cancelButton = new JButton("CANCEL");
okButton     = new JButton("OK");
```

The text we pass to the constructor is the label of a button. After the buttons are created, we must place them on the frame's content pane.

There are two general approaches to placing buttons (and other types of GUI objects) on a frame's content pane, one that uses a layout manager and another that does not. The ***layout manager*** for a container is an object that controls the placement of the GUI objects. For example, the simplest layout manager called **FlowLayout** places GUI objects in the top-to-bottom, left-to-right order. If we do not use any layout manager, then we place GUI objects by explicitly specifying their position and size on the content pane. We call this approach ***absolute positioning***.

In this section,we will use **FlowLayout**. To use the flow layout, we set the layout manager of a frame's content pane by passing an instance of **FlowLayout** to the **setLayout** method:

```
contentPane.setLayout(new FlowLayout());
```

After the layout manager is set, we add the two buttons to the content pane, so they become visible when the frame is displayed on the screen:

```
                contentPane.add(okButton);
                contentPane.add(cancelButton);
```

example:

```
import javax.swing.*;
import java.awt.*;
class JButtonFrame extends JFrame {
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 200;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
private JButton cancelButton;
private JButton okButton;
public JButtonFrame() {
Container contentPane = getContentPane( );
//set the frame properties
setSize     (FRAME_WIDTH, FRAME_HEIGHT);
setResizable(false);
setTitle    ("Program JButtonFrame");
setLocation (FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
//set the layout manager
contentPane.setLayout(new FlowLayout());
//create and place two buttons on the frame's content pane
okButton = new JButton("OK");
contentPane.add(okButton);
cancelButton = new JButton("CANCEL");
contentPane.add(cancelButton);
//register 'Exit upon closing' as a default close operation
setDefaultCloseOperation(EXIT_ON_CLOSE);
} }
}
public static void main(String[] args){
JButtonFrame frame = new JButtonFrame();
frame.setVisible(true);
}
```

event-driven programming:

To build an effective graphical user interface using objects from the **javax.swing** and **java.awt** packages, we must learn a new style of program control called ***event driven programming***.

An *event* occurs when the user interacts with a GUI object.
For example, when you move the cursor, click on a button, or select a menu choice, an event occurs. In event-driven programs, we program objects to respond to these events by defining event-handling methods. we will learn the fundamentals of event-driven programming. Since the main objective is to teach the fundamentals of GUI and event-driven programming and not to provide an exhaustive coverage of the Swing classes, we will cover only the most common GUI objects.

Handling Button Events:

Now let's study how we process the button clicks. An action such as clicking a button is called an ***event***, and the mechanism to process the events *event handling*. event handling is implemented by two types of objects:**event source objects** and **event listener objects**.

A GUI object, such as a button, where the event occurs is called an *event*, or simply, the ***event source***. We say an event source *generates* events. So, for example, when the user clicks on a button, the corresponding **JButton** object will generate an action event. When an event is generated, the system notifies the relevant event listener objects.
An *event listener object*, or simply an ***event listener***, is an object that includes a method that gets executed in response to generated events. It is possible for a single object to be both an event source and an event listener.

- ❖ Among the many different types of events, the most common one is called an ***action event***. For example, when a button is clicked or a menu item is selected, an event source will generate an action event. For the generated events to be processed, we must associate, or register, event listeners to the event sources. If the event sources have no registered listeners, then generated events are simply ignored (this is what happened in the **JButtonFrame** program). For each type of event, we have a corresponding listener. For example, we have action listeners for action events, window listeners for window events, mouse listeners for mouse events, and so forth. Event types other than action events are discussed later . If we wish to process the action events generated by a button, then we must associate an action listener to the button.

An object that can be registered as an action listener must be an instance of a class that is declared specifically for the purpose. We call such class an *action listener class*. For this sample program, let's name the action listener class **ButtonHandler**.
We will describe how to define the **ButtonHandler** class shortly. But first we will show the step to register an instance of **ButtonHandler** as the

action listener of the two action event sources—**okButton** and **cancelButton**—of the sample frame window.
An action listener is associated to an action event source by calling the event source's **addActionListener** method with this action listener as its argument. For example, to register an instance of **ButtonHandler** as an action listener of **okButton** and **cancelButton**, we can execute the following code:

```
ButtonHandler handler = new ButtonHandler( );
okButton.addActionListener(handler);
cancelButton.addActionListener(handler);
```

Notice that we are associating a single **ButtonHandler** object as an action listener of both buttons, because, although we can, it is not necessary to associate two separate listeners, one for the **OK** button and another for the **CANCEL** button. A single listener can be associated to multiple event sources. Likewise, although not frequently used, multiple listeners can be associated to a single event source.

When an event source generates an event, the system checks for matching registered listeners (e.g., for action events the system looks for registered action listeners, for window events the system looks for registered window listeners, and so forth). If there is no matching listener, the event is ignored. If there is a matching listener, the system nofies the listener by calling the listener's corresponding method. In case of action events, this method is **actionPerformed**. To ensure that the programmer includes the necessary **actionPerformed** method in the action listener class, the class must implement the **ActionListener** interface. The **ButtonHandler** class, for example, must be defined in the following way:

```
class ButtonHandler implements ActionListener {
//data members and constructors come here
public void actionPerformed(ActionEvent evt){
}
//event-handling statements come here
}
```

An argument to the **actionPerformed** method is an **ActionEvent** object that represents an action event, and the **ActionEvent** class includes methods to access the properties of a generated event. We want to change the title of a frame to **You clicked OK** or **You clicked CANCEL** depending on which button is clicked. This is done inside the **actionPerformed** method. The general idea of the method is as follows:

```
public void actionPerformed(ActionEvent evt){
String buttonText= get the text of the event source;
JFrame frame= the frame that contains this event source;
frame.setTitle("You clicked " + buttonText);
}
```

The first statement retrieves the text of the event source (the text of the **okButton** is the string **OK** and the text of the **cancelButton** is the string **CANCEL**). We can do this in two ways.

The first way is to use the **getActionCommand** method of the action event object **evt**. Using this method, we can retrieve the text of the clicked button as

String buttonText = evt.getActionCommand();

The second way is to use the **getSource** method of the action event object **evt**. Using this method, we can retrieve the text of the clicked button as

```
JButton clickedButton = (JButton) evt.getSource();
String  buttonText    = clickedButton.getText();
```

```
/* Displays a frame with two buttons and associates an instance of
ButtonHandler to the two buttons*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class JButtonEvents extends JFrame {
private static final int FRAME_WIDTH  = 300;
private static final int FRAME_HEIGHT = 200;
class ButtonHandler implements ActionListener {
  public ButtonHandler() {
    public void actionPerformed(ActionEvent event){
     JButton clickedButton = (JButton) event.getSource();
    String  buttonText = clickedButton.getText();
    frame.setTitle("You clicked " + buttonText);
   }
  }}
```

Or:

```
class JButtonFrameHandler extends JFrame implements ActionListener
{
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 200;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
private JButton cancelButton;
private JButton okButton;
public JButtonFrameHandler() {
Container c = getContentPane( );
//set the frame properties
setSize     (FRAME_WIDTH, FRAME_HEIGHT);
setResizable(false);
setTitle    ("Program JButtonFrameHandler");
setLocation (FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
//set the layout manager
c.setLayout(new FlowLayout());
//create and place two buttons on the frame's content pane
okButton = new JButton("OK");
c.add(okButton);
cancelButton = new JButton("CANCEL");
c.add(cancelButton);
//register this frame as an action listener of the two buttons
cancelButton.addActionListener(this);
okButton.addActionListener(this);
//register 'Exit upon closing' as a default close operation
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent event){
JButton clickedButton = (JButton) event.getSource();
String buttonText = clickedButton.getText();
setTitle("You clicked " + buttonText);
}}
public static void main(String[] args){
JButtonFrameHandler frame = new JButtonFrameHandler();
frame.setVisible(true);
}
```

Notice how we call the **addActionListener** method of **cancelButton** and **okButton**. This frame object is the action event listener, so we pass it as an argument to the method as

```
   cancelButton.addActionListener(this);
   okButton.addActionListener(this);
```

Likewise, because the **actionPerformed** method now belongs to this frame class itself, we can call other methods of the frame class from the **actionPerformed** method without dot notation. So the statement to change the title is simply

```
    setTitle("You clicked " + buttonText);
```

Text-Related GUI Components

In this section we will introduce three Swing GUI classes—**JLabel**, **JTextField**, **JPasswordField** and **JTextArea**—that deal with text. The first two deal with a single line of text and the last with multiple lines of text. A **TextField** object allows the user to enter a single Like a **JButton** object, an instance of **JTextField and JPasswordField** generates an action events. A **JTextField** object generates an action event when the user presses the Enter key while the object is active (it is active when you see the vertical blinking line in it).

JLabel, on the other hand, does not generate any event. A **JTextArea** object also generates events, specfiically the types of events called *text events* and *document events*. Handling of these events is more involved than handling action events, so to keep the discussion manageable, we won't be processing the **JTextArea** events.

We will describe the **JTextField** class first. We set a **JTextField** object's size and position and register its action listener in the same way as we did for the **JButton** class. To illustrate its use, we will modify the **JButtonFrameHandler** by adding a single **JTextField** object. We will call the new class **TextFrame1**. The effect of clicking the buttons **CANCEL** and **OK** is the same as before. If the user presses the Enter key while the **JTextField** object is active, then we will change the title to whatever text is entered in this **JTextField** object. In the data declaration part, we add JTextField inputLine; and in the constructor we create a **JTextField** object and register the frame as its action listener:

```
public TextFrame1 {
...
inputLine = new JTextField();
inputLine.setColumns(22);
add(inputLine);
}
inputLine.addActionListener(this);
...
```

Notice the use of **setColumns** method instead of **setSize** in the earlier examples. We do not use the **setSize** method to set the size of a textfield. The number we pass to the **setColumns** method does not necessarily mean the number of characters visible on the text field because the default font may be a variable-pitch font. If we set the font to a fixed-pitch font as in:

```
    inputLine.setColumns(20);
    inputLine.setFont(new Font("Courier", Font.PLAIN, 14));
```

then 20 characters will be visible. Also, notice that the **setColumns** method affects the number of characters visible by setting the size of the text field. It does not affect the number of characters we can enter in the text field. There is no fixed boundline of text, while a **JLabel** object is for displaying uneditable text. on the number of characters we can enter. When we enter more than the visible number of characters, then the text will scroll to the left.

```
/* Displays a frame with two buttons, one text field and two labels*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class TextFrame2 extends JFrame implements ActionListener {
...
private JLabel prompt;
private JLabel image;
public TextFrame2() {
. . .
image = new JLabel(new ImageIcon("cat.gif"));
image.setSize(50, 50);
contentPane.add(image);
prompt = new JLabel();
prompt.setText("Please enter your name");
prompt.setSize(150, 25);
contentPane.add(prompt);
...}
...}
public static void main(String[] args){
TextFrame2 frame = new TextFrame2();
frame.setVisible(true);
}
```

Now we need to modify the **actionPerformed** method to handle both the button click events and the Enter key events. We have three event sources (two buttons and one textfield), so the first thing we must do in the **actionPerformed** method is to determine the source. We will use the **instanceof** operator to determine the class to which the event source belongs. Here's the general idea:

```
if (event.getSource() instanceof JButton){
//event source is either cancelButton
//or okButton
...
} else { //event source must be inputLine
...}
```

We use the **getText** method of **JTextField** to retrieve the text that the user has entered. The complete method is written as

```
public void actionPerformed(ActionEvent event)
{
   if (event.getSource() instanceof JButton){
   JButton clickedButton = (JButton) event.getSource();
   String buttonText = clickedButton.getText();
   setTitle("You clicked " + buttonText);
  } else { //the event source is inputLine
     setTitle("You entered '" + inputLine.getText() + "'");
}}
```

- ✓ Notice that we can—but did not—write the **else** part as

```
  JTextField textField = (JTextField) event.getSource();
  setTitle("You entered '" + textField.getText() + "'");
```

because we know that the event source is **inputLine** in the **else** part. So we wrote it more succinctly as

```
   setTitle("You entered '" + inputLine.getText() + "'");
```

Another approach to event handling is to associate a **ButtonHandler** to the two button event sources and a **TextHandler** (need to add this new class) to the textfield event source.

Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class TextFrame1 extends JFrame implements ActionListener {
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 200;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
private JButton cancelButton;
private JButton okButton;
private JTextField inputLine;
public TextFrame1() {
Container contentPane;
//set the frame properties
setSize     (FRAME_WIDTH, FRAME_HEIGHT);
setResizable(false);
setTitle    ("Program Ch14SecondJFrame");
setLocation (FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
contentPane = getContentPane();
contentPane.setLayout( new FlowLayout());
inputLine = new JTextField( );
inputLine.setColumns(22);
contentPane.add(inputLine);
inputLine.addActionListener(this);
//create and place two buttons on the frame
okButton = new JButton ("OK");
contentPane.add(okButton);
cancelButton = new JButton ("CANCEL");
contentPane.add(cancelButton);
//register this frame as an action listener of the two buttons
cancelButton.addActionListener(this);
okButton.addActionListener(this);
//register 'Exit upon closing' as a default close operation
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent event){
if (event.getSource() instanceof JButton){
JButton clickedButton = (JButton) event.getSource();
String buttonText = clickedButton.getText();
setTitle("You clicked " + buttonText);
} else { //the event source is inputLine
setTitle("You entered '" + inputLine.getText() + "'");
}}
public static void main(String[] args){
TextFrame1 frame = new TextFrame1();
frame.setVisible(true);
}
```

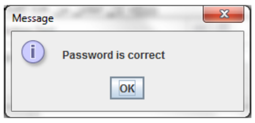
<u>JPasswordField :</u>

EXAMPLE:

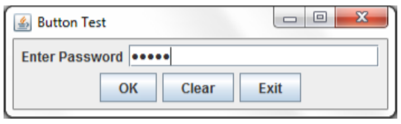
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class ButtonTest extends JFrame implements
ActionListener
{
   JPasswordField password;
   JButton ok, clear, exit;
   public ButtonTest()
   {
      super("Button Test");
      Container c = getContentPane();
      c.setLayout(new FlowLayout());
      c.add(new JLabel("Enter Password"));
```

```
        password = new JPasswordField(20);
        c.add(password);
        ok = new JButton("OK");
        ok.addActionListener(this);
        c.add(ok);
        clear = new JButton("Clear");
        clear.addActionListener(this);
        c.add(clear);
        exit = new JButton("Exit");
        exit.addActionListener(this);
        c.add(exit);
        setSize(350, 100);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent event )
    {
        if (event.getSource() == exit)
            System.exit(0);
        else
        if (event.getSource() == clear)
            password.setText(" ");
        else
        { // ok
            if(password.getText().equals("Sudan"))
                JOptionPane.showMessageDialog(this, "Password is correct")
            else
                JOptionPane.showMessageDialog(this, "Sorry..wrong password");
        }
    }
    public static void main(String args[])
    { new ButtonTest();}
   }
```

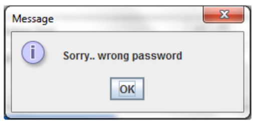
Output:



```
Message
Password is correct
OK
```



```
Button Test
Enter Password •••••
OK  Clear  Exit
```



```
Message
Sorry.. wrong password
OK
```

TextArea:

Now let's create the example by using a **JTextArea** object. We will call the sample class **TextFrame3**. In this sample program, we will add two buttons labeled **ADD** and **CLEAR**, one text field, and one text area to a frame. When a text is entered in the text field and the Enter (Return) key is pressed or the **ADD** button is clicked, the entered text is added to the list shown in the text area. Figure down shows the state of this frame after six words are entered.

We declare a **JTextArea** object **textArea** in the data member section as **private** JTextArea textArea; and add the statements to create it inside the constructor as:

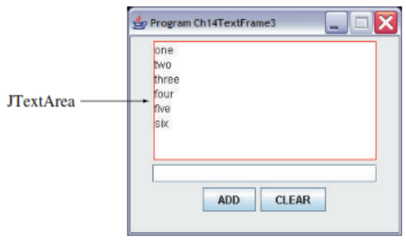
```
textArea = new JTextArea();
textArea.setColumns(22);
textArea.setRows(8);
textArea.setBorder(BorderFactory.createLineBorder(Color.RED));
textArea.setEditable(false);
contentPane.add(textArea);
```

By default, unlike the single-line **JTextField**, the rectangle that indicates the boundary of a **JTextArea** object is not displayed on the frame. We need to create the border for a **JTextArea** object explicitly. The easiest way to do so is to call one of the class methods of the **BorderFactory** class. In the example, we called the **createLineBorder** method with a **Color** object as its argument. We passed **Color.RED** so the red rectangle is displayed. The **createLineBorder** method returns a properly created **Border** object, and we pass this **Border** object to the **setBorder** method of the text area object. There are other interesting borders you might want to try. The API documentation of the **BorderFactory** class records more options and variations. In the sample frame, we do not want the user to edit the text displayed in the text area, so we disable editing by the statement textArea.setEditable(**false**);

Example:

```
/*Displays a frame with two buttons, one text field, and one text area*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class TextFrame3 extends JFrame implements ActionListener {
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 250;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
private static final String EMPTY_STRING = "";
private static final String NEWLINE =
System.getProperty("line.separator");
private JButton    clearButton;
private JButton    addButton;
private JTextField inputLine;
private JTextArea  textArea;
public TextFrame3() {
Container contentPane;
//set the frame properties
setSize     (FRAME_WIDTH, FRAME_HEIGHT);
setResizable(false);
setTitle    ("Program TextFrame3");
setLocation (FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
contentPane = getContentPane();
contentPane.setLayout(new FlowLayout ());
textArea = new JTextArea();
textArea.setColumns(22);
textArea.setRows(8);
textArea.setBorder(BorderFactory.createLineBorder(Color.RED));
textArea.setEditable(false);
contentPane.add(textArea);
inputLine = new JTextField();
inputLine.setColumns(22);
contentPane.add(inputLine);
inputLine.addActionListener(this);
//create and place two buttons on the frame
addButton = new JButton ("ADD");
contentPane.add(addButton);
clearButton = new JButton ("CLEAR");
contentPane.add(clearButton);
//register this frame as an action listener of the two buttons
clearButton.addActionListener(this);
addButton.addActionListener(this);
//register 'Exit upon closing' as a default close operation
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent event){
if (event.getSource() instanceof JButton){
JButton clickedButton = (JButton) event.getSource();
if (clickedButton == addButton){
```

```
addText(inputLine.getText());
} else {
clearText();
}
} else { //the event source is inputLine
addText(inputLine.getText());
}}
private void addText(String newline){
textArea.append(newline + NEWLINE);
inputLine.setText("");
}
private void clearText() {
textArea.setText(EMPTY_STRING);
inputLine.setText(EMPTY_STRING); }}
 public static void main(String[] args){
TextFrame3 frame = new TextFrame3();
frame.setVisible(true);
}
```



```
one
two
three
four
five
six
```

To add a text to the text area, we use the **append** method. Notice that we cannot use the **setText** method of **JTextArea** here because it will replace the old content with the new text. What we want here is to add new text to the current content. Also, since we need to add new text on a separate line, we need to output the new-line control character **\n**. Here's the basic idea for adding new text to the text area object **textArea**:

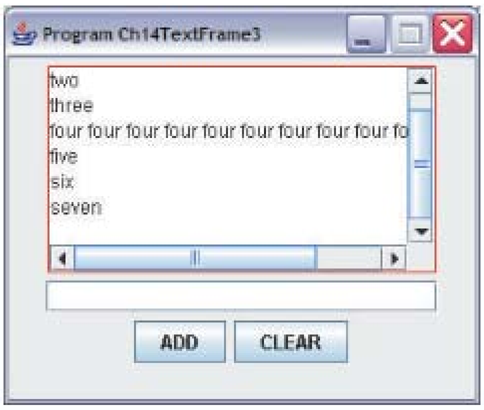
```
   String enteredText = inputLine.getText();
   textArea.append(enteredText + "\n");
```

Because the actual sequence of characters to separate lines is dependent on the operating systems, if we want to maintain consistent behavior across all operating systems, it is best to not use afixed character such as **\n**. Instead, we should call the **getProperty** method of the **System** class, passing the string **line.separator** as anargument, to get the actual sequence of characters used by the operating system on which the program is being executed. We can define a class constant as :

```
Private static final String NEWLINE=System.getProperty("line.separator");
```

and use it in the program as:

```
textArea.append(enteredText + NEWLINE);
```



```
two
three
four four four four four four four four fo
five
six
seven
```

Using a JScrollPane to Add Scroll Bars Automatically

When we run the **TextFrame3** class and add more rows (lines) of text than the number of rows set by calling the **setRows** method, what happens? The height of the text area gets taller. Likewise, the text area expands horizontally when we enter a line longer than the specified width. This is not a desired behavior. The easiest way to handle the situation is to wrap the text area with an instance of **javax.swing.JScrollPane** that adds the vertical and horizontal scroll bars when necessary.

In the original **TextFrame3** class, this is what we did to create and set the **JTextArea** object:

```
textArea = new JTextArea();
textArea.setColumns(22);
textArea.setRows(8);
textArea.setBorder(BorderFactory.createLineBorder(Color.RED));
textArea.setEditable(false);
contentPane.add(textArea);
```

To add scroll bars that will appear automatically when needed, we replace the last statement above with the following:

```
JScrollPane scrollText= new JScrollPane(textArea);
scrollText.setSize(200, 135);
contentPane.add(scrollText);
```

Notice that the properties, such as the border and bounds, of the **JScrollPane** object are set, no longer the properties of the **JTextArea**. Figure shows a sample **TextFrame3** object when the **JScrollPane** class is used.

JComboBox

The **JComboBox** class presents a combo box. This class is similar to the **JRadioButton** class in that it also allows the user to select one item from a list of possible choices. The difference between the two lies in how the choices are presented to the user. Another name for a combo box is a *drop-down list*, which is more descriptive of its interaction style.
We can construct a new **JComboBox** by passing an array of **String** objects, for example,

```
String[] comboBoxItem= {"Java", "C++", "Smalltalk", "Ada"};
JComboBox comboBox = new JComboBox(comboBoxItem);
```

A **JComboBox** object generates both action events and item events. An action event is generated every time a **JComboBox** is clicked (note it is not that common to process action events of **JComboBox**). Every time an item different from the currently selected item is selected, an item event is generated and the **itemStateChanged** method is called twice. Thefirst time is for the deselection of the currently selected item, and the second is for the selection of the new item. Notice that when the same item is selected again, no item event is generated.
To find out the currently selected item, we call the **getSelectedItem** method of **JComboBox**. Because the return type of this method is **Object**, we must typecast to the correct type. For this example, items are **String** objects, so we write

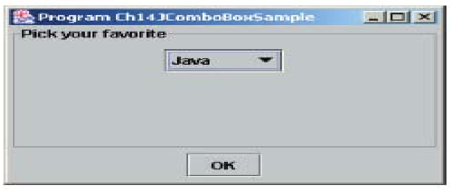
```
String selection = (String) comboBox.getSelectedItem();
```

Also, we can call the **getSelectedIndex** method to retrieve the position of the selected item. The first item in the list is at position 0.
Here’s the **JComboBoxSample** class:

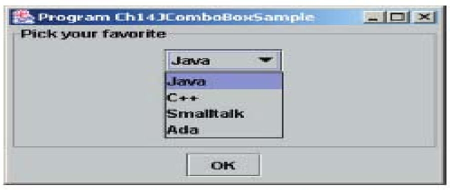
Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class JComboBoxSample extends JFrame implements ActionListener,
ItemListener   {
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 200;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
private JComboBox comboBox;
public JComboBoxSample() {
Container contentPane;
JPanel    comboPanel, okPanel;
JButton   okButton;
String[] comboBoxItem = {"Java", "C++", "Smalltalk", "Ada"};
//set the frame properties
setSize    (FRAME_WIDTH, FRAME_HEIGHT);
setTitle   ("Program JComboBoxSample");
setLocation(FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
contentPane = getContentPane( );
contentPane.setBackground(Color.WHITE);
contentPane.setLayout(new BorderLayout());
//create and place a combo box
comboPanel = new JPanel(new FlowLayout());
comboPanel.setBorder(BorderFactory.createTitledBorder("Pick your favorite"));
comboBox = new JComboBox(comboBoxItem);
comboBox.addItemListener(this);
comboPanel.add(comboBox);
//create and place the OK button
okPanel = new JPanel(new FlowLayout());
okButton = new JButton("OK");
okButton.addActionListener(this);
okPanel.add(okButton);
contentPane.add(comboPanel, BorderLayout.CENTER);
contentPane.add(okPanel, BorderLayout.SOUTH);
//register 'Exit upon closing' as a default close operation
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent event){
String favorite;
int loc;
favorite = (String) comboBox.getSelectedItem();
loc      = comboBox.getSelectedIndex();
JOptionPane.showMessageDialog(this, "Currently selected item '" +
favorite + "' is at index position " + loc);
}
public void itemStateChanged(ItemEvent event){
   String state;
   if (event.getStateChange() == ItemEvent.SELECTED){
     state = "is selected ";}
  else {
    state = "is deselected ";}
JOptionPane.showMessageDialog(this, "JComboBox Item '" +
event.getItem() +
"' " + state);
}}
}
```

```
public static void main(String[] args){
JComboBoxSample frame = new JComboBoxSample();
frame.setVisible(true);
}
```



The state when the frame first appeared on the screen.



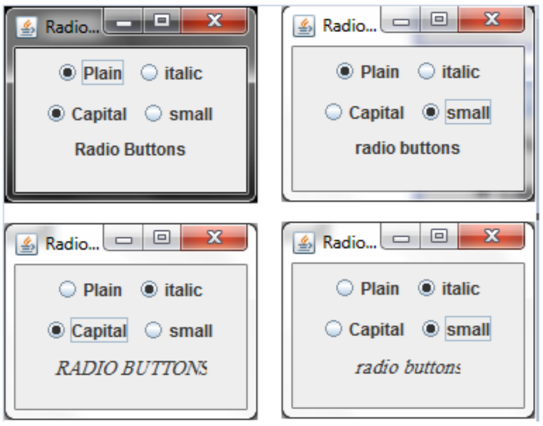
The state after the items in the combo box are revealed by clicking on the down arrow.

RadioButton

Example:

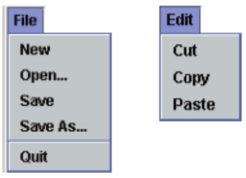
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class RadioButtonTest extends JFrame implements ItemListener
{
   JLabel text;
   JRadioButton plain, bold, italic, capital, small;
   ButtonGroup styleGroup , caseGroup;
   public RadioButtonTest()
   {
      super("Radio Buttons");
      Container c= getContentPane();
      c.setLayout(new FlowLayout());
      styleGroup = new ButtonGroup();
      plain = new JRadioButton("Plain", true);
      plain. addItemListener(this);
      styleGroup.add(plain);
      c.add(plain);
      bold = new JRadioButton("Bold", false);
      bold.addItemListener(this);
      styleGroup.add(bold);
      italic = new JRadioButton("italic", false);
      italic.addItemListener(this);
      styleGroup.add(italic);
      c.add(italic);
      caseGroup = new ButtonGroup();
```

```
        capital = new JRadioButton("Capital", true);
        capital.addItemListener(this);
        caseGroup.add(capital);
        c.add(capital);
        small = new JRadioButton("small", false);
        small.addItemListener(this);
        caseGroup.add(small);
        c.add(small);
        text = new JLabel("Radio Buttons");
        c.add(text);
        setSize(180, 140);
        setVisible(true);
    }
    public void itemStateChanged(ItemEvent event)
    {
        if(event.getSource() == plain)
            text.setFont(new Font("serif", Font.PLAIN, 14));
        if (event.getSource() == bold)
            text.setFont(new Font("serif", Font.BOLD, 14));
        if (event.getSource() == italic)
            text.setFont(new Font("serif", Font.ITALIC, 14));
        if (event.getSource() == capital)
            text.setText("RADIO BUTTONS");
        if (event.getSource() == small)
            text.setText("radio buttons");
    }
    public static void main(String args[])
    {
        new RadioButtonTest();
    }
}
```

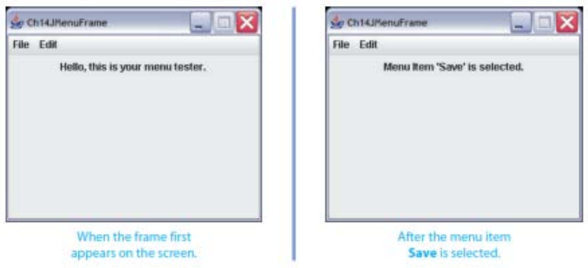


menu

Practical programs with a graphical user interface will almost always support menus. In this section we will describe how to display menus and process menu events by using **JMenu**, **JMenuItem**, and **JMenuBar** from the **javax.swing** package.
Let's write a sample code to illustrate the display of menus and the processing of menu item selections. We will create two menus, **File** and **Edit**, with the following menu items:



If the menu item **Quit** is selected, then we terminate the program. When a menu item other than **Quit** is selected, we print a message that idenfies the selected menu item, for example, Menu item 'New' is selected Figure shows a **JMenuFrame** when it isfirst opened and after the menu choice **Save** is selected.
One possible sequence of steps to create and add menus is this:
1. Create a **JMenuBar** object and attach it to a frame.
2. Create a **JMenu** object.
3. Create **JMenuItem** objects and add them to the **JMenu** object.
4. Attach the **JMenu** object to the **JMenuBar** object.


Ch14JMenuFrame
File Edit
Hello, this is your menu tester.
When the frame first appears on the screen.
Ch14JMenuFrame
File Edit
Menu item 'Save' is selected.
After the menu item Save is selected.

Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class JMenuFrame extends JFrame implements ActionListener {
private static final int FRAME_WIDTH    = 300;
private static final int FRAME_HEIGHT   = 250;
private static final int FRAME_X_ORIGIN = 150;
private static final int FRAME_Y_ORIGIN = 250;
private JLabel response;
private JMenu  fileMenu;
private JMenu  editMenu;
public JMenuFrame(){
Container contentPane;
//set the frame properties
setTitle    ("JMenuFrame");
setSize     (FRAME_WIDTH, FRAME_HEIGHT);
setResizable(false);
setLocation (FRAME_X_ORIGIN, FRAME_Y_ORIGIN);
contentPane = getContentPane( );
contentPane.setLayout(new FlowLayout());
//create two menus and their menu items
createFileMenu();
createEditMenu();
//and add them to the menu bar
JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);
menuBar.add(fileMenu);
menuBar.add(editMenu);
//create and position response label
response = new JLabel("Hello, this is your menu tester.");
response.setSize(250, 50);
contentPane.add(response);
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent event){
String menuName;
menuName = event.getActionCommand();
if (menuName.equals("Quit")) {
System.exit(0);
} else {
response.setText("Menu Item '" + menuName + "' is selected.");
}
}
private void createFileMenu( ) {
JMenuItem item;
fileMenu = new JMenu("File");
item = new JMenuItem("New"); //New
item.addActionListener(this);
fileMenu.add(item);
item = new JMenuItem("Open"); //Open...
item.addActionListener(this);
fileMenu.add(item);
item = new JMenuItem("Save"); //Save
item.addActionListener(this);
fileMenu.add(item);
item = new JMenuItem("Save As..."); //Save As...
item.addActionListener(this);
fileMenu.add(item);
fileMenu.addSeparator();        //add a horizontal separator line
item = new JMenuItem("Quit"); //Quit
item.addActionListener(this);
fileMenu.add(item);
}
}}

private void createEditMenu() {
JMenuItem item;
editMenu = new JMenu("Edit");
item = new JMenuItem("Cut"); //Cut
item.addActionListener(this);
editMenu.add(item);
item = new JMenuItem("Copy"); //Copy
item.addActionListener(this);
editMenu.add(item);
item = new JMenuItem("Paste"); //Paste
item.addActionListener(this);
editMenu.add(item);
}}
public static void main(String[] args){
JMenuFrame frame = new JMenuFrame();
frame.setVisible(true);
}
```

If the size of text for the response label is too small, then we can make it bigger by including the following statement in the constructor:

```
response.setFont( new Font("Helvetica", /*font name*/ Font.BOLD,   /*font style*/16 ) );    /*font size*/
```

Generic:

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.
Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

What Are Generics?

At its core, the term generics means parameterized types. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called generic, as in generic class or generic method.
It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type Object. Because Object is the superclass of all other classes, an Object reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used Object references to operate on various types of objects. The problem was that they could not do so with type safety.
Generics add the type safety that was lacking. They also streamline the process, because it is no longer necessary to explicitly employ casts to translate between **Object** and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit. Thus, generics expand your ability to reuse code and let you do so safely and easily.

Generics Work Only with Objects:

When declaring an instance of a generic type, the type argument passed to the type parameter must be a class type. You cannot use a primitive type, such as **int** or **char**. For example, with **Gen**, it is possible to pass any class type to **T**, but you cannot pass a primitive type to a type parameter. Therefore, the following declaration is illegal:

```
Gen<int> strOb = new Gen<int>(53); // Error, can't use primitive type
```

Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers (as the preceding example did) to encapsulate a primitive type.
Further, Java’s autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

❖ Generic Methods:

You can write a *single generic method* declaration that can be called with *arguments of different types*. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

☐ All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

☐ Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

☐ The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

☐ A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example:

Following example illustrates how we can print an array of different type using a single Generic method:

```
public class GenericMethodTest
{ // generic method printArray
  public static < E > void printArray( E[] inputArray )
  {// Display array elements
      for ( E element : inputArray ){
        System.out.printf( "%s ", element );
      }
      System.out.println();
   }
   public static void main( String args[] )
   {// Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
```

```
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
}
```

output:

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

❖ Generic Classes:

Let’s begin with a simple example of a generic class. The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.
The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

class *class-name*<*type-param-list*> { // ...

Here is the syntax for declaring a reference to a generic class:

***class-name*<*type-arg-list*>*var-name*=new *class-name*<*type-arg-list*>(*cons-arg-list*);**

Example:

```
// A simple generic class. Here, T is a type parameter that will be replaced
by a real type when an object of type Gen is created.
class Gen<T> {
  T ob;
// declare an object of type T Pass the constructor a reference to an object of type T.
Gen(T o) {
  ob = o;}
 // Return ob.
T getob() {
  return ob;}
// Show type of T.
void showType() {
  System.out.println("Type of T is " +ob.getClass().getName());
// Demonstrate the generic class.
 }}
class GenDemo {
 public static void main(String args[]) {
  // Create a Gen reference for Integers.
  Gen<Integer> iOb;
  /* Create a Gen<Integer> object and assign its reference to iOb.  Notice the use of
autoboxing to encapsulate the value 88 within an Integer object.*/
  iOb = new Gen<Integer>(88);
  // Show the type of data used by iOb.
  iOb.showType();
  // Get the value in iOb. Notice that no cast is needed.
  int v = iOb.getob();
  System.out.println("value: " + v);
  System.out.println();
// Create a Gen object for Strings.
 Gen<String> strOb = new Gen<String>("Generics Test");
// Show the type of data used by strOb.
  strOb.showType();
// Get the value of strOb. Again, notice
// that no cast is needed.
String str = strOb.getob();
System.out.println("value: " + str);
}}
```

output:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

Let’s examine this program carefully.
First, notice how **Gen** is declared by the following line:

```
class Gen<T> {
```

Here, **T** is the name of a *type parameter*. This name is used as a placeholder for the actual type that will be passed to **Gen** when an object is created. Thus, **T** is used within **Gen** whenever the type parameter is needed. Notice that **T** is contained within <>. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets. Because **Gen** uses a type parameter, **Gen** is a generic class, which is also called a *parameterized type.*

How Generics Improve Type Safety?

At this point, you might be asking yourself the following question: Given that the same functionality found in the generic **Gen** class can be achieved without generics, by simply specifying **Object** as the data type and employing the proper casts, what is the benefit of making **Gen** generic? The answer is that generics automatically ensure يضمن the type safety of all operations involving **Gen**. In the process, they eliminateيهمل the need for you to enter casts and to type-check code by hand.
To understand the benefits of generics, first consider the following program that creates a non-generic equivalent متساويof **Gen**:

```
// NonGen is functionally equivalent to Gen but does not use generics.
class NonGen {
Object ob; /* ob is now of type Object Pass the constructor a reference to an object
of type Object*/
NonGen(Object o) {
ob = o;
}
// Return type Object.
Object getob() {
return ob; }
// Show type of ob.
void showType() {
System.out.println("Type of ob is " +ob.getClass().getName());
// Demonstrate the non-generic class.
}}
class NonGenDemo {
public static void main(String args[]) {
NonGen iOb;
// Create NonGen Object and store an Integer in it. Autoboxing still occurs.
iOb = new NonGen(88);
// Show the type of data used by iOb.
iOb.showType();
// Get the value of iOb.
// This time, a cast is necessary.
int v = (Integer) iOb.getob();
System.out.println("value: " + v);
System.out.println();
// Create another NonGen object and
// store a String in it.
NonGen strOb = new NonGen("Non-Generics Test");
// Show the type of data used by strOb.
strOb.showType();
// Get the value of strOb.
// Again, notice that a cast is necessary.
String str = (String) strOb.getob();
System.out.println("value: " + str);
// This compiles, but is conceptually wrong!
iOb = strOb;
v = (Integer) iOb.getob(); // run-time error!
}}
```

There are several things of interest in this version. First, notice that **NonGen** replaces all uses of **T** with **Object**. This makes **NonGen** able to store any type of object, as can the generic version. However, it also prevents the Java compiler from having any real knowledge about the type of data actually stored in **NonGen**, which is bad for two reasons. First, explicit casts must be employed to retrieve the stored data.
Second, many kinds of type mismatch errors cannot be found until run time. Let's look closely at each problem.

Example:
Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

```
public class MaximumTest
{
  // determines the largest of three Comparable objects
  public static <T extends Comparable<T>> T maximum(T x, T y, T z)
  {
    T max = x; // assume x is initially the largest
    if ( y.compareTo( max ) > 0 ){
      max = y; // y is the largest so far
    }
    if ( z.compareTo( max ) > 0 ){
      max = z; // z is the largest now
    }
    return max; // returns the largest object
  }
  public static void main( String args[] )
  {
    System.out.printf( "Max of %d, %d and %d is %d\n\n",
             3, 4, 5, maximum( 3, 4, 5 ) );
    System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
             6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
    System.out.printf( "Max of %s, %s and %s is %s\n","pear",
      "apple", "orange", maximum( "pear", "apple", "orange" ) );
  }
}
```

output:

```
maximum of 3, 4 and 5 is 5
maximum of 6.6, 8.8 and 7.7 is 8.8
maximum of pear, apple and orange is pear
```

Bounded Type Parameters:

In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter. For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles. Thus, you want to specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like this:

```
// Stats attempts (unsuccessfully) to create a generic class that can compute
// the average of an array of numbers of any given type.
// The class contains an error!
class Stats<T> {
T[] nums; // nums is an array of type T
// Pass the constructor a reference to an array of type T.
Stats(T[] o) {
nums = o;}
// Return type double in all cases.
double average() {
double sum = 0.0;
for(int i=0; i < nums.length; i++)
sum += nums[i].doubleValue(); // Error!!!
return sum / nums.length;
}}
```