CHAPTER 13

Managing Errors and Exceptions

13.1 INTRODUCTION

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing to program to produce unexpected results. *Errors* are the wrongs that can make a program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

13.2 TYPES OF ERRORS

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

Compile-Time Errors

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the **.class** file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

Program 13.1 *Illustration of compile-time errors*

```
/* This program contains an error */
class Error1
{
      public static void main(String args[ ])
      {
            System.out.println("Hello Java!") // Missing;
      }
}
```

The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in Program 13.1, the following message will be displayed in the screen:

```
Error1.java :7: ';' expected
System.out.println ("Hello Java!")
^
1 error
```

We can now go to the appropriate line, correct the error, and recompile the program. Sometimes, a single error may be the source of multiple errors later in the compilation. For example, use of an undeclared variable in a number of places will cause a series of errors of type "undefined variable". We should generally consider the earliest errors as the major source of our problem. After we fix such an error, we should recompile the program and look for other errors.

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character. The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments / initialization
- Bad references to objects
- Use of = in place of = = operator
- And so on

Other errors we may encounter are related to directory paths. An error such as

```
javac : command not found
```

means that we have not set the path correctly. We must ensure that the path includes the directory where the Java exécutables are stored.

Run-Time Errors

Sometimes, a program may compile successfully creating the **.class** file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string
- And may more

When such errors are encountered, Java typically generates an error message and aborts the program. Program 13.2 illustrates how a run-time error causes termination of execution of the program.

Program 13.2 *Illustration of run-time errors*

```
class Error2
{
    public static void main(String args[ ])
    {
        int  a  =  10;
        int  b  =  5;
        int  c  =  5;
        int x = a/(b-c);          // Division by zero
        System.out.println("x = " + x);
        int y = a/(b+c);
        System, out. println("y = " + y);
    }
}
```

Program 13.2 is syntactically correct and therefore does not cause any problem during compilation. However, while executing, it displays the following message and stops without executing further statements.

```
java.lang.ArithmeticException: / by zero
    at Error2.main(Error2.java:10)
```

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

13.3 EXCEPTIONS

An *exception* is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e. informs us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message as shown in the output of Program 13.2 and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling.*

The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

1. Find the problem (***Hit*** the exception).
2. Inform that an error has occurred (***Throw*** the exception)
3. Receive the error information (***Catch*** the exception)
4. Take corrective actions (***Handle*** the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exceptions that we must watch out for catching are listed in Table 13.1.

Table 13.1 *Common Java Exceptions*

Exception Type	*Cause of Exception*
`ArithmeticException`	Caused by math errors such as division by zero
`ArrayIndexOutOfBoundsException`	Caused by bad array indexes
`ArrayStoreException`	Caused when a program tries to store the wrong type of data in an array
`FileNotFoundException`	Caused by an attempt to access a nonexistent file
`IOException`	Caused by general I/O failures, such as inability to read from a file
`NullPointerException`	Caused by referencing a null object
`NumberFormatException`	Caused when a conversion between strings and number fails
`OutOfMemoryException`	Caused when there's not enough memory to allocate a new object
`SecurityException`	Caused when an applet tries to perform an action not allowed by the browser's security setting
`StackOverFlowException`	Caused when the system runs out of stack space
`StringIndexOutOfBoundsException`	Caused when a program attempts to access a nonexistent character position in a string

Exceptions in Java can be categorised into two types:

- **Checked exceptions:** These exceptions are explicitly handled in the code itself with the help of try-catch blocks. Checked exceptions are extended from the **java.lang.Exception** class.
- **Unchecked exceptions:** These exceptions are not essentially handled in the program code; instead the JVM handles such exceptions. Unchecked exceptions are extended from the **java.lang.RuntimeException** class.

It is important to note that checked and unchecked exceptions are absolutely similar as far as their functionality is concerned; the difference lies only in the way they are handled.

13.4 SYNTAX OF EXCEPTION HANDLING CODE

The basic concepts of exception handling are throwing an exception and catching it. This is illustrated in Fig. 13.1.

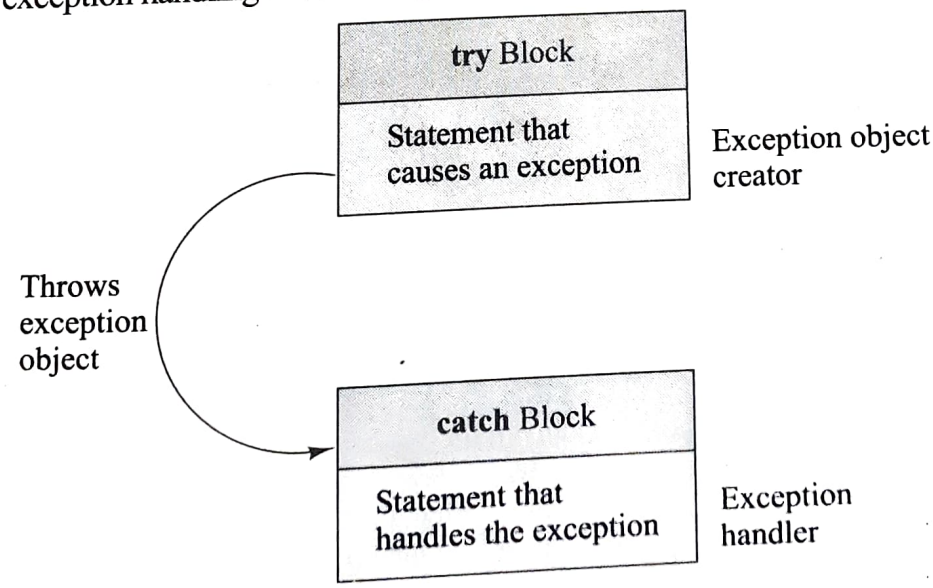


Fig. 13.1 *Exception handling mechanism*

Java uses a keyword **try** to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword **catch** "catches" the exception "thrown" by thee try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple **try** and **catch** statements:

```
.....
.....
try
{
     statement;     // generates an exception
}
catch (Exception-type e)
{
     statement;     // processes the exception
}
.....
.....
```

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception. Remember that every **try** statement should be followed by *at least one* **catch** statement; otherwise compilation error will occur.

Note that the **catch** statement works like a method definition. The **catch** statement is passed a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

Program 13.3 illustrates the use of try and catch blocks to handle an arithmetic exception. Note that Program 13.3 is a modified version of Program 13.2.

Program 13.3 *Using try and catch for exception handling*

```
class Error3
{
     public static void main(String args[ ])
     {
          int a = 10;
          int b = 5;
          int c = 5;
          int x, y ;
          try
          {
               x = a / (b-c);     // Exception here
          }
          catch (ArithmeticException e)
          {
               System.out.println("Division by zero");
          }
          y = a / (b+c);
          System, out. println("y = " + y);
     }
}
```

Program 13.3 displays the following output:

```
Division by zero
y = 1
```

Note that the program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and then continues the execution, as if nothing has happened. Compare with the output of Program 13.2 which did not give the value of **y**.

Program 13.4 shows another example of using exception handling mechanism. Here, the try-catch block catches the invalid entries in the list of command line arguments.

Program 13.4 *Catching invalid command line arguments*

```
class CLinelnput
{
    public static void main(String args[ ])
    {
        int  invalid = 0; // Number of invalid arguments
        int  number, count = 0;
        for  (int i = 0; i < args.length; i++)
        {
            try
            {
                number = Integer.parselnt(args[i]);
            }
            catch(NumberFormatException e)
            {
               invalid = invalid + 1;   // Caught an invalid number
               System.out.println("Invaiid Number: "+args[i]);
               continue ;               // Skip the remaining part of the loop
            }
            count = count + 1;
        }
        System.out.println("Valid Numbers = " + count);
        System.out.println("Invalid Numbers = " + invalid);
    }
}
```

Note the use of the wrapper class **Integer** to obtain an **int** number from a string:

```
number = Integer.parselnt(args[i])
```

Remember that the numbers are supplied to the program through the command line and therefore they are stored as strings in the array **args[ ]**. Since the above statement is placed in the try block, an exception is thrown if the string is improperly formatted and the number is not included in the count.

When we run the program with the command line:

```
java CLinelnput 15 25.75 40 Java 10.5 65
```

it produces the following output:

```
Invalid Number:  25.75
Invalid Number:   Java
Invalid Number:   10.5
Valid Numbers      = 3
Invalid Numbers    = 3
```

There could be situations where there is a possibility of generation of multiple exceptions of different types within a particular block of the program code. We can use nested try statements in such situations. The execution of the corresponding catch blocks of nested try statements is done using a stack. The Program 13.5 shows the example of nested try statements:

Program 13.5 *Nested try statements*

```
class eg_nested_try
{
    public static void main(String args[])
    {
        try
        {
            int a=2,b=4,c=2,x=7,z ;
            int p[ ]={2};
            p[3]=33;
            try
            {
                z=x/((b*b)-(4*a*c));
                    System.out.println("The value of z is = "+ z );
            }
                catch(ArithmeticException e)
                {
                    System.out.println("Division by zero in Arithmetic
                    expression");
                }
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index is out-of-bounds ");
        }
    }
}
```

Program 13.5 displays the following output:

```
Array index is out-of-bounds
```

13.5 MULTIPLE CATCH STATEMENTS

It is possible to have more than one catch statement in the catch block as illustrated below:

```
.........
.........
try
{
    statement ;                    // generates an exception
}
catch (Exception-Type-1 e)
}
    statement;                     // processes exception type 1

}
catch (Exception-Type-2 e)
```

```
{
    statement;                    // processes exception type 2
}
.
.
.
catch (Exception-Type-N e)
{
    statement ;                   // processes exception type N
}
.........
.........
```

When an exception in a **try** block is generated, the Java treats the multiple **catch** statements like cases in a **switch** statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

Note that Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

Example:

```
catch (Exception e);
```

The **catch** statement simply ends with a semicolon, which does nothing. This statement will catch an exception and then ignore it.

Program 13.6 *Using multiple catch blocks*

```
class Erro4
{
    public static void main(String args[ ])
    {
        int a[ ] = {5, 10);
        int b = 5;
        try
        {
            int x = a[2] / b - a[1];
        }
        catch(ArithmeticException e)
        }
            System.out.println "(Division by zero");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index error");
        }
        catch(ArrayStoreException e)
        {
            System.out.print!n("Wrong data type");
        }
        int y = a[1] / a[0];
        System, out. println("y = " + y);
    }
```

Program 13.6 uses a chain of catch blocks and, when run, produces the following output:

```
Array index error
y = 2
```

Note that the array element a[2] does not exist because array **a** is defined to have only two elements, a[0] and a[1]. Therefore, the index 2 is outside the array boundary thus causing the block

```
Catch(ArrayIndexOutOfBoundsException e)
```

to catch and handle the error. Remaining catch blocks are skipped.

13.6 USING FINALLY STATEMENT

Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements, **finally** block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows:

```
try
{
     ..........
     ..........
}
finally
{
     ..........
     ..........
}
```

```
try
{
     ..........
     ..........
}
catch  (....)
{
     ..........
     ..........
}
catch  (....)
}
     ..........
     ..........
}
.
.
.
finally
}
     ..........
     ..........
}
```

When a **finally** block is defined, this is guaranteed to execute, regardless of whether or not in exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

In Program 13.6, we may include the last two statements inside a **finally** block as shown below:

```
finally
{
     int y = a[1]/a[0];
     System.out.println("y = "  +y);
}
```

This will produce the same output.

13.7 THROWING OUR OWN EXCEPTIONS

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows:

```
throw new Throwable subclass;
```

Examples:

```
throw new ArithmeticException( );
throw new NumberFormatException( );
```

Program 13.7 demonstrates the use of a user-defined subclass of Throwable class. Note that **Exception** is a subclass of **Throwable** and therefore **MyException** is a subclass of **Throwable** class. An object of a class that extends **Throwable** can be thrown and caught.

Program 13.7 *Throwing our own exception*

```
import java.lang.Exception;
class MyException extends Exception
{
    MyException(String message)
    {
        super(message);
    }
}
class TestMyException
{
    public static void main(Strings args[ ])
    {
        int x = 5, y = 1000;
        try
        {
            float z = (float) x / (float) y ;
            if(z < 0.01)
            {
                throw new MyException("Number is too small");
            }
        }
        catch (MyException e)
        }
            System.out.println("Caught my exception");
            System.out.println(e.getMessage( ) );
        }
        finally
        {
            System.out.println ("I am always here");
        }
    }
}
```

A run of Program 13.7 produces:

```
Caught my exception
Number is too small
I am always here
```

The object e which contains the error message "Number is too small" is caught by the **catch** block which then displays the message using the getMessage( ) method.

Note that Program 13.7 also illustrates the use of **finally** block. The last line of output is produced by the **finally** block.

There could be situations where there is a possibility that a method might throw certain kinds of exceptions but there is no exception handling mechanism prevalent within the method. In such a case, it is important that the method caller is intimated explicitly that certain types of exceptions could be expected from the called method, and the caller must get prepared with some catching mechanism to deal with it.

The **throws** clause is used in such a situation. It is specified immediately after the method declaration statement and just before the opening brace. The Program 13.8 shows an example of using the throws clause:

Program 13.8 *Use of throws*

```
class Examplethrows
{
static void divide_m() throws ArithmeticException
{
int x = 22,y= 0,z;
z = x/y;
}
public static void main(String args[])
{
try
{
divide_m( );
}
catch(ArithmeticException e)
{
System.out.println("Caught the exception " + e);
}
}
}
```

Program 13.8 displays the following output:

```
Caught the exception java.lang.ArithmeticException: / by zero
```

13.8 USING EXCEPTIONS FOR DEBUGGING

As we have seen, the exception-handling mechanism can be used to hide errors from rest of the program. It is possible that the programmers may misuse this technique for hiding errors rather than debugging the code. Exception handling mechanism may be effectively used to locate the type and place of errors. Once we identify the errors, we must try to find out why these errors occur before we cover them up with exception handlers.

13.9 SUMMARY

A good program does not produce unexpected results. We should incorporate features that could check for potential problem spots in programs and guard against program failures. The problem conditions known as exceptions in Java must be handled carefully to avoid any program failures.

In this chapter we have discussed the following:

- What exceptions are
- How to throw system exceptions
- How to define our own exceptions
- How to catch and handle different types of exceptions
- Where to use exception handling tools

We must ensure that common exceptions are handled where appropriate.

KEY TERMS

Errors, Compile-time errors, Run-time errors, Exception, Exception handling, try, catch, throw, finally.

REVIEW QUESTIONS

13.1 What is an exception?
13.2 How do we define a **try** block?
13.3 How do we define a **catch** block?
13.4 List some of the most common types of exceptions that might occur in Java. Give examples.
13.5 Is it essential to catch all types of exceptions?
13.6 How many **catch** blocks can we use with one **try** block?
13.7 Create a **try** block that is likely to generate three types of exception and then incorporate necessary **catch** blocks to catch and handle them appropriately.
13.8 What is a **finally** block? When and how is it used? Give a suitable example.
13.9 Explain how exception handling mechanism can be used for debugging a program.
13.10 Define an exception called "NoMatchException" that is thrown when a string is not equal to "India". Write a program that uses this exception.

DEBUGGING EXERCISES

13.1 The following code catches Arithmetic Exception. Will this code work? If not, why?

```
class excep1
{
    public static void main(String args[])
    {
        try
        {
            int n=Integer.parseint(args[0]);
            int n1=Integer.parseInt(args[1]);
            int n2=n + n1;
        }
        catch(ArithmeticException ex)
```

```
        {
            System.out.println("Arithmetic Exception block 1");
        }
        catch(ArithmeticException ex)
        {
            System.out.println("Arithmetic Exception block 2");
        }
    }
}
```

13.2 Which exception may be thrown if the given code is executed?

```
class excep2
{
    public static void main(String args[])
    {
        try
        {
            int n = Integer.parselnt(args[0]);
            int nl = Integer.parseint(args[l]);
            int n2 = n+nl;
            System.out.pri ntln("Sum is " + n2);
        }
        catch(ArithmeticException ex)
        {
            System.out.println("ArithmeticException:"+ex.getMessage());
        }
        catch(NumberFormatException ex)
        {
            System.out.println("Format Exception : "+ex.getMessage());
        }
        catch(Exception ex)
        {
            System.out.pri ntln("Exception :" + ex);
        }
    }
}
```

13.3 The code throws an exception in case the desired input is not received. Debug the code.

```
class excep3
{
    public static void main(String args[])
    {
        if(args[0]=="Hello")
            System.out.println("String is right");
        else
            throw new Exception("Invaiid String");
    }
}
```

13.4 Custom exception has been created in the code given below. Correct the code.

```
class myexception extends Exception
{
    myexception(String s)
    {
        super(s);
    }
}
```

```
class excep4
{
    public static void main(String args[])
    {
        if(args[0]=="Hello")
            System.out.println("String is right");
        else
        try
        {
            throw new myexception("Invalid String");
        }catch(myexception ex)
        {
            System.out.println(ex.getmessage());
        }
    }
}
```

13.5 The program calculates sum of two numbers inputted as command-line arguments. When will it give an exception?

```
class excep
{
    public static void main(String []args)
    {
        Try {
            int n = Integer.parseInt(args[0]);
            int n1 = Integer.parseInt(args[1]);
            int n2 = n+n1;
            System.out.println("Sum is: " + n2);
            }
        catch(NumberFormatException ex)
        {
            System.out.println(ex);
        }
    }
}
```

CHAPTER 14

Applet Programming

14.1 INTRODUCTION

Applets are small Java programs that are primarily used in Internet computing. They can be transported over the Internet from one computer to another and run using the **Applet Viewer** or any Web browser that supports Java. An applet, like any application program, can do many things for us. It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation, and play interactive games.

Java has revolutionized the way the Internet users retrieve and use documents on the world wide network. Java has enabled them to create and use fully interactive multimedia Web documents. A web page can now contain not only a simple text or a static image but also a Java applet which, when run, can produce graphics, sounds and moving images. Java applets therefore have begun to make a significant impact on the World Wide Web.

Local and Remote Applets

We can embed applets into Web pages in two ways. One, we can write our own applets and embed them into Web pages. Second, we can download an applet from a remote computer system and then embed it into a Web page.

An applet developed locally and stored in a local system is known as a *local applet.* When a Web page is trying to find a local applet, it does not need to use the Internet and therefore the local system does not require the Internet connection. It simply searches the directories in the local system and locates and loads the specified applet (see Fig. 14.1).

A *remote applet* is that which is developed by someone else and stored on a remote computer

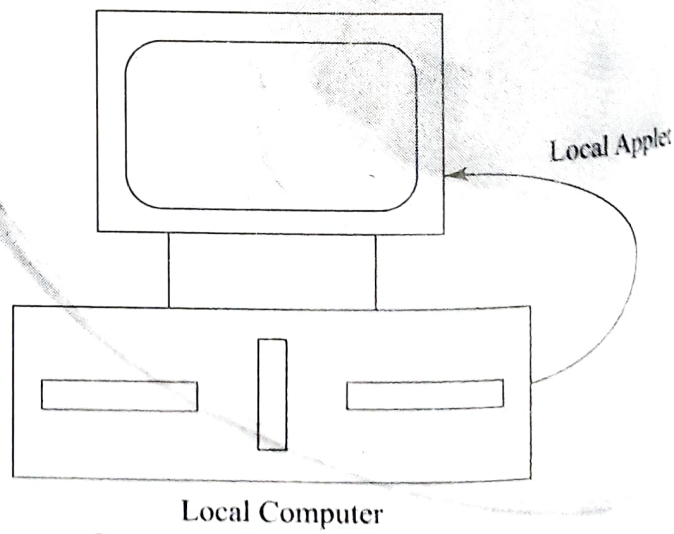


Fig. 14.1 *Loading local applets*

connected to the Internet. If our system is connected to the Internet, we can download the remote applet onto our system via at the Internet and run it (see Fig. 14.2).

In order to locate and load a remote applet, we must know the applet's address on the Web. This address is known as *Uniform Resource Locator (URL)* and must be specified in the applet's HTML document as the value of the CODEBASE attribute (see Section 14.11). Example:

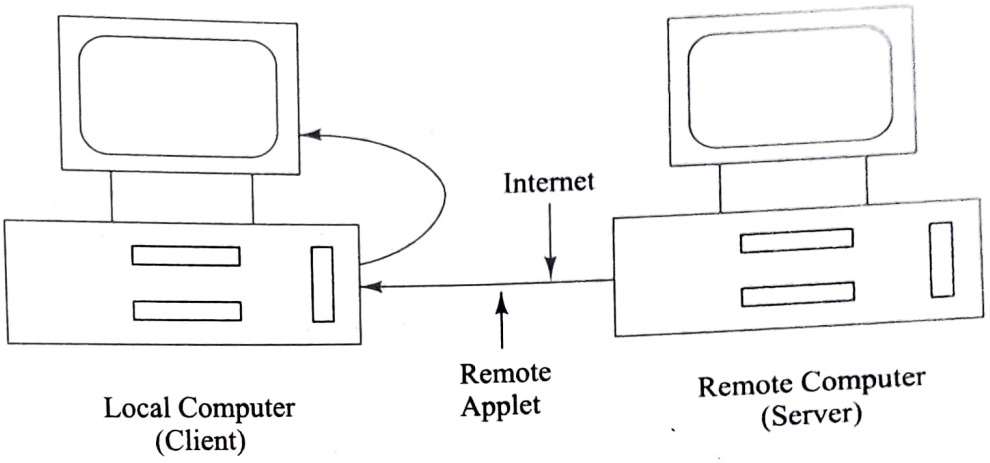


Fig. 14.2 *Loading a remote applet*

```
CODEBASE = http : // www,netserve.com / applets
```

In the case of local applets, CODEBASE may be absent or may specify a local directory. In this chapter we shall discuss how applets are created, how they are located in the Web documents and how they are loaded and run in the local computer.

14.2 HOW APPLETS DIFFER FROM APPLICATIONS

Although both the applets and stand-alone applications are Java programs, there are significant differences between them. Applets are not full-featured application programs. They are usually written to accomplish a small task or a component of a task. Since they are usually designed for use on the Internet, they impose certain limitations and restrictions in their design.

- Applets do not use the **main( )** method for initiating the execution of the code. Applets, when loaded, automatically call certain methods of Applet class to start and execute the applet code.
- Unlike stand-alone applications, applets cannot be run independently. They are run from inside a Web page using a special feature known as HTML tag.
- Applets cannot read from or write to the files in the local computer.
- Applets cannot communicate with other servers on the network.
- Applets cannot run any program from the local computer.
- Applets are restricted from using libraries from other languages such as C or C++. (Remember, Java language supports this feature through **native** methods).

All these restrictions and limitations are placed in the interest of security of systems. These restrictions ensure that an applet cannot do any damage to the local system.

14.3 PREPARING TO WRITE APPLETS

Until now, we have been creating simple Java application programs with a single **main( )** method that created objects, set instance variables and ran methods. Here, we will be creating applets exclusively and therefore we will need to know

- When to use applets,
- How an applets works,
- What sort of features an applet has, and
- Where to start when we first create our own applets.

First of all, let us consider the situations when we might need to use applets.

1. When we need something dynamic to be included in the display of a Web page. For example, an applet that displays daily sensitivity index would be useful on a page that lists share prices of various companies or an applet that displays a bar chart would add value to a page that contains data tables.
2. When we require some "flash" outputs. For example, applets that produce sounds, animations or some special effects would be useful when displaying certain pages.
3. When we want to create a program and make it available on the Internet for us by others on their computers.

Before we try to write applets, we must make sure that Java is installed properly and also ensure that either the Java **appletviewer** or a Java-enabled browser is available. The steps involved in developing and testing in applet are:

1. Building an applet code (**.java** file)
2. Creating an executable applet (**.class** file)
3. Designing a Web page using HTML tags
4. Preparing <APPLET> tag
5. Incorporating <APPLET> tag into the Web page
6. Creating HTML file
7. Testing the applet code

Each of these steps is discussed in the following sections.

14.4 BUILDING APPLET CODE

It is essential that our applet code uses the services of two classes, namely, **Applet** and **Graphics** from the Java class library. The **Applet** class which is contained in the **java.applet** package provides life and behaviour to the applet through its methods such as **init( ), start( )** and **point( )**. Unlike the applications, where Java calls the **main( )** method directly to initiate the execution of the program, when an applet is loaded, Java automatically calls a series of **Applet** class methods for starting, running, and stopping the applet code. The **Applet** class therefore maintains the *lifecycle* of an applet.

The **paint( )** method of the **Applet** class, when it is called, actually displays the result of the applet code on the screen. The output may be text, **graphics**, or sound. The **paint( )** method, which requires a **Graphics** object as an argument, is defined as follows:

```
public void paint (Graphics g)
```

This requires that the applet code imports the **java.awt** package that contains the **Graphics** class. All output operations of an applet are performed using the methods defined in the **Graphics** class. It is thus clear from the above discussions that an applet code will have a general format as shown below:

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

. . . . . .
. . . . . .
```

```
public class appletclassname extends Applet
{
     ..............
     ..............
     public void paint (Graphics g)
{
          ..............
          ..............
          ..............                    // Applet operations code
     }
     ..............
     ..............
}
```

The *appletclassname* is the main class for the applet. When the applet is loaded, Java creates an instance of this class, and then a series of **Applet** class methods are called on that instance to execute the code. Program 14.1 shows a simple HelloJava applet.

Program 14.1 *The HelloJava applet*

```
import java.awt.*;
import java.applet.*;
public class HelloJava extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString "Hello  Java",  10,  100);
    }
}
```

The applet contains only one executable statement.

```
g.drawString("Hello Java", 10, 100);
```

which, when executed, draws the string

Hello Java

at the position 10, 100 (pixels) of the applet's reserved space as shown in Fig. 14.3.

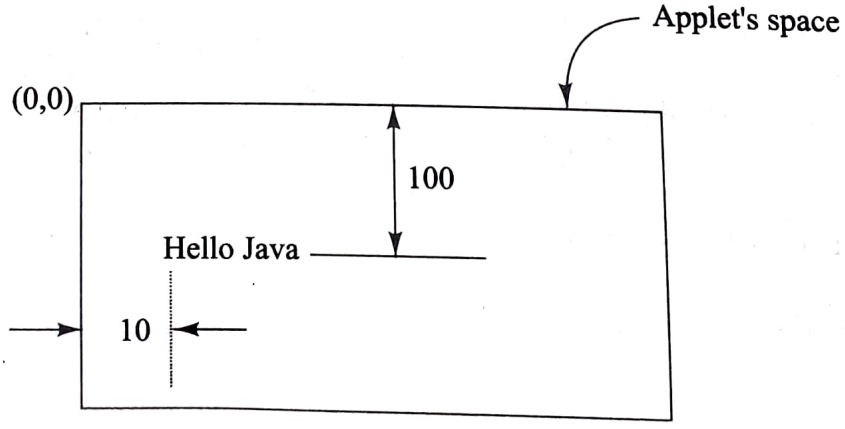


Fig. 14.3 *Output of Program 14.1*

Remember that the applet code in Program 14.1 should be saved with the file name **Hello Java. java,** a java subdirectory. Note the **public** keyword for the class **HelloJava.** Java requires that the main applet ass be declared public.

Remember that **Applet** class itself is a subclass of the **Panel** class, which is again a subclass of the **Container** class and so on as shown in Fig. 14.4. This shows that the main applet class inherits properties from a long chain of classes. An applet can, therefore, use variables and methods from all these classes.

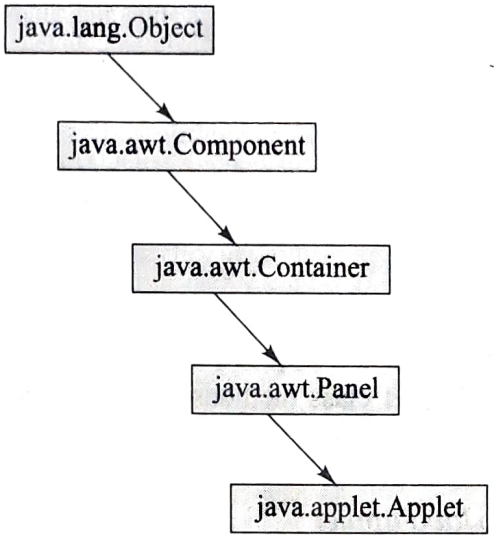


Fig. 14.4 *Chain of classes inherited by Applet class*

14.5 APPLET LIFE CYCLE

Every Java applet inherits a set of default behaviours from the **Applet** class. As a result, when an applet is loaded, it undergoes a series of changes in its state as shown in Fig. 14.5. The applet states include:

- Born on initialization state
- Running state
- Idle state
- Dead or destroyed state

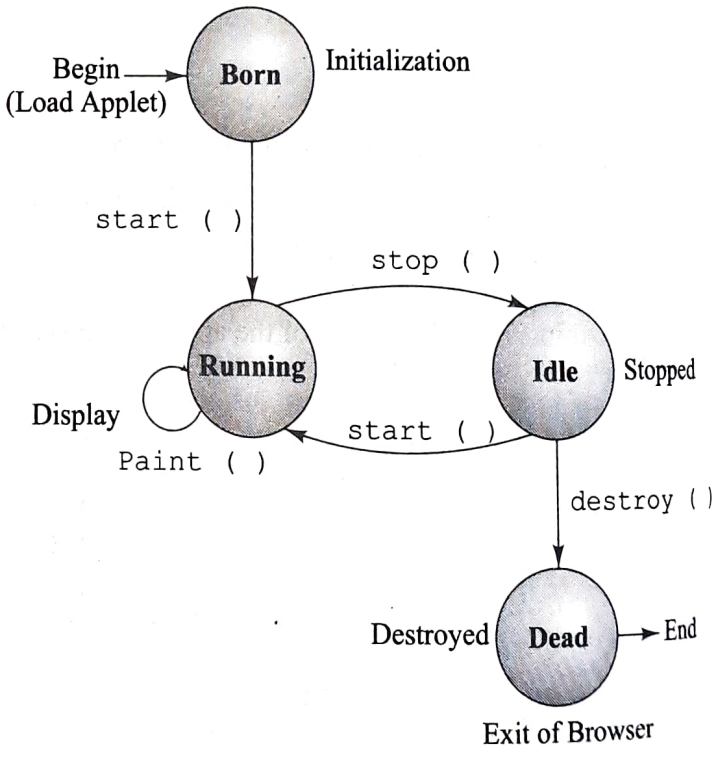


Fig. 14.5 *An applet's state transition diagram*

Initialization State

Applet enters the *initialization* state when it is first loaded. This is achieved by calling the **init( )** method of **Applet** Class. The applet is born. At this stage, we may do the following, if required.

- Create objects needed by the applet
- Set up initial values
- Load images or fonts
- Set up colors

The initialization occurs only once in the applet's life cycle. To provide any of the behaviours mentioned above, we must override the **init( )** method:

```
public void init(  )
{
     ............
     ............ (Action)
}
```

Running State

Applet enters the ***running*** state when the system calls the **start( )** method of **Applet** Class. This occurs automatically after the applet is initialized. Starting can also occur if the applet is already in 'stopped" (idle) state. For example, we may leave the Web page containing the applet temporarily to another page and return back to the page. This again starts the applet running. Note that, unlike **init( )** method, the **start( )** method may be called more than once. We may override the **start( )** method to create a thread to control the applet.

```
public voids start( )
{
     ............
     ............
     ............   (Action)
}
```

Idle or Stopped State

An applet becomes *idle* when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling the **stop( )** method explicitly. If we use a thread to run the applet, then we must use **stop( )** method to terminate the thread. We can achieve this by overriding the **stop( )** method;

```
public void stop(  )
{
     ............
     ............    (Action)
     ............
}
```

Dead State

An applet is said to be *dead* when it is removed from memory. This occurs automatically by invoking the **destroy( )** method when we quit the browser. Like initialization, destroying stage occurs only once in the applet's life cycle. If the applet has created any resources, like threads, we may override the **destroy( )** method to clean up these resources.

```
public void destroy ( )
{
     ............
     ............ (Action)
     ............
}
```

Display State

Applet moves to the *display* state whenever it has to perform some output operations on the screen. This happens immediately after the applet enters into the running state. The **paint( )** method is called to

accomplish this task. Almost every applet will have a **paint( )** method. Like other methods in the life cycle, the default version of **paint( )** method does absolutely nothing. We must therefore override this method if we want anything to be displayed on the screen.

```
public void paint (Graphics g)
{
     ..............
     ..............     (Display statements)
     ..............
}
```

It is to be noted that the display state is not considered as a part of the applet's life cycle. In fact, the **paint( )** method is defined in the **Applet** class. It is inherited from the **Component** class, a super class of **Applet.**

14.6 CREATING AN EXECUTABLE APPLET

Executable applet is nothing but the **.class** file of the applet, which is obtained by compiling the source code of the applet. Compiling an applet is exactly the same as compiling an application. Therefore, we can use the Java compiler to compile the applet.

Let us consider the **HelloJava** applet created in Section 14.4. This applet has been stored in a file called **HelloJava.java.** Here are the steps required for compiling the **HelloJava** applet.

1. Move to the directory containing the source code and type the following command:

   ```
   javac HelloJava.java
   ```

2. The compiled output file called **HelloJava.class** is placed in the same directory as the source.
3. If any error message is received, then we must check for errors, correct them and compile the applet again.

14.7 DESIGNING A WEB PAGE

Recall the Java applets are programs that reside on Web pages. In order to run a Java applet, it is first necessary to have a Web page that references that applet.

A Web page is basically made up of text and HTML tags that can be interpreted by a Web browser or an applet viewer. Like Java source code, it can be prepared using any ASCII text editor. A Web page is also known as HTML page or HTML document. Web pages are stored using a file extension. **html** such as **MyApplet.html.** Such files are referred to as HTML files. HTML files should be stored in the same directory as the compiled code of the applets.

As pointed out earlier, Web pages include both text that we want to display and HTML tags (commands) to Web browsers. A Web page is marked by an opening HTML tag < HTML> and a closing HTML tag </HTML> and is divided into the following three major sections:

1. Comment section (Optional)
2. Head section (Optional)
3. Body section

A Web page outline containing these three sections and the opening and closing HTML tags is illustrated in Fig. 14.6.

Comment Section

This section contains comments about the Web page. It is important to include comments that tell us what is going on in the Web page. A comment line begins with a <! and ends with a >. Web browsers will ignore the text enclosed between them. Although comments are important, they should be kept to a minimum as they will be downloaded along with the applet. Note that comments are optional and can be included anywhere in the Web page.

Head Section

The head section is defined with a starting <HEAD> tag and a closing </HEAD> tag. This section usually contains a title for the Web page as shown below:

```
<HEAD>
        <TITLE> Welcome to Java Applets </TITLE>
</HEAD>
```

The text enclosed in the tags <TITLE> and </TITLE> will appear in the title bar of the Web browser when it displays the page. *The head section is also optional.*

Note that tags < .... > containing HTML commands usually appear impairs such as <HEAD> and </HEAD>, and <TITLE> and </TITLE>. A slash (/) in a tag signifies the end of that tag section.

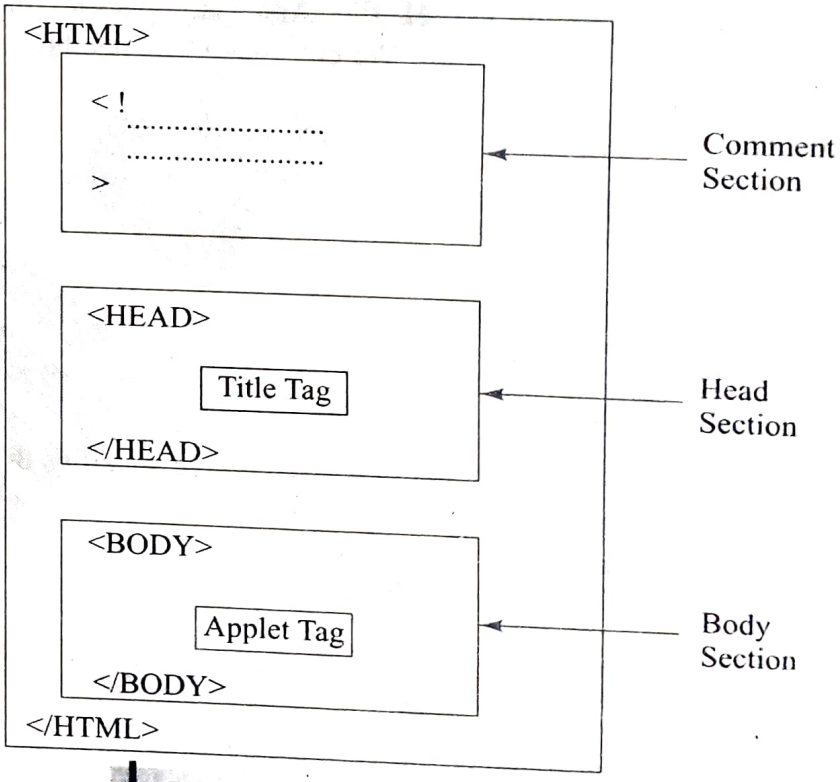


Fig. 14.6 *A Web page template*

Body Section

After the head section comes the body section. We call this as body section because this section contains the entire information about the Web page and its behaviour. We can set up many options to indicate how our page must appear on the screen (like colour, location, sound, etc.,). Shown below is a simple body section:

```
<BODY>
    <CENTER>
        <H1> Welcome to the World of Applets </H1>
    </CENTER>
    <BR>
    <APPLET ...>
    </APPLET>
</BODY>
```

The body shown above contains instructions to display the message

```
Welcome to the World of Applets
```

followed by the applet output on the screen. Note that the <CENTER> tag makes sure that the text is centered and <H1> tag causes the text to be of the largest size. We may use other heading tags <H2> to <H6> to reduce the size of letters in the text.

14.8 APPLET TAG

Note that we have included a pair of <APPLET...> and </APPLET> tags in the body section discussed above. The <APPLET ...> tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires. The ellipsis in the tag < APPLET ...> indicates that it contains certain attributes that must specified. The <APPLET> tag given below specifies the minimum requirements to place the **HelloJava** applet on a Web page:

```
<APPLET
     CODE = helloJava.class
     WIDTH = 400
     HEIGHT = 200 >
</APPLET >
```

This HTML code tells the browser to load the compiled Java applet **HelloJava.class,** which is in the same directory as the HTML file. And also specifies the display area for the applet output as 400 pixels width and 200 pixels height. We can make this display area appear in the centre of the screen by using the CENTER tags shown as follows:

```
<CENTER>
     <APPLET
     ..........

     ..........

     ..........
     </APPLET>
</CENTER>
```

Note that <APPLET> tag discussed above specifies three things:

1. Name of the applet
2. Width of the applet (in pixels)
3. Height of the applet (in pixels)

14.9 ADDING APPLET TO HTML FILE

Now we can put together the various components of the Web page and create a file known as HTML file. Insert the <APPLET> tag in the page at the place where the output of the applet must appear. Following is the content of the HTML file that is embedded with the <APPLET> tag of our **HelloJava** applet.

```
<HTML>
   <! This page includes a welcome title in the title bar and also
   displays a welcome message. Then it specifies the applet to be loaded
   and executed.

   >
   <HEAD>
      <TITLE>
           Welcome to Java Applets
      </TITLE>
   </HEAD>
   <BODY>
```

```
    <CENTER>
        <H1> Welcome to the World of Applets </H1>
</CENTER>
<BR>
<CENTER>
    <APPLET
        CODE = HelloJava.class
        WIDTH = 400
        HEIGHT = 200>
    </APPLET>
</CENTER>
</BODY>
</HTML>
```

We must name this file as **HelloJava.html** and save it in the same directory as the compiled applet.

14.10 RUNNING THE APPLET

Now that we have created applet files as well as the HTML file containing the applet, we must have the following files in our current directory:

```
HelIoJava.java
HelIoJava.class
HelIoJava.html
```

To run an applet, we require one of the following tools:

1. Java-enabled Web browser (such as HotJava or Netscape)
2. Java appletviewer

If we use a Java-enabled Web browser, we will be able to see the entire Web page containing the applet. If we use the **appletviewer** tool, we will only see the applet output. Remember that the **appletviewer** is not a full-fledged Web browser and therefore it ignores all of the HTML tags except the part pertaining to the running of the applet.

The **appletviewer** is available as a part of the Java Development Kit that we have been using so far. We can use it to run our applet as follows:

```
appletviewer HelloJava.html
```

Notice that the argument of the **appletviewer** is not the **.java** file or the **.class** file, but rather **.html** file. The output of our applet will be as shown in Fig. 14.7.

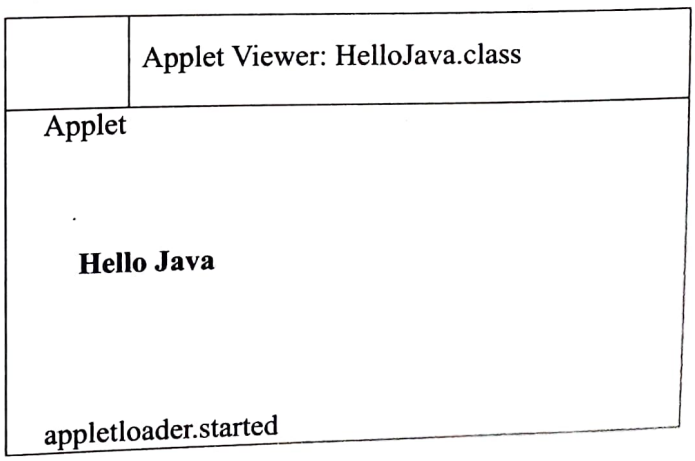


Fig. 14.7 *Output of HelloJava applet by using appletviewer*

14.11 MORE ABOUT APPLET TAG

We have used the <APPLET> tag in its simplest form. In its simplest form, it merely creates a space of the required size and then displays the applet output in that space. The syntax of the <APPLET> tag is a little

more complex and includes several attributes that can help us better integrate our applet into the overall design of the Web page. The syntax of the <APPLET> tag in full form is shown as follows:

```
<APPLET
     [ CODEBASE = codebase_URL ]
     CODE = AppletFileName.class
     [ ALT = alternate_text ]
     [ NAME = applet_instance_name ]
     WIDTH = pixels
     HEIGHT = pixels
     [ ALIGN = alignment ]
     [ VSPACE = pixels ]
     [ HSPACE = pixels ]
>
[ < PARAM NAME = name1 VALUE = value1> ]
[ < PARAM NAME = name2 VALUE = value2> ]
.............
.............
[ Text to be displayed in the absence of Java ]
</APPLET>
```

The various attributes shown inside [ ] indicate the options that can be used when integrating an applet into a Web page. Note that the minimum required attributes are:

```
CODE = AppletFileName.class
WIDTH = pixels
HEIGHT = pixels
```

Table 14.1 lists all the attributes and their meaning.

Table 14.1 *Attributes of APPLET Tag*

Attribute	*Meaning*
CODE=AppletFileName.class	Specifies the name of the applet class to be loaded. That is, the name of the already-compiled .class file in which the executable Java bytecode for the applet is stored. This attribute must be specified.
CODEBASE=codebase_URL (Optional)	Specifies the URL of the directory in which the applet resides. If the applet resides in the same directory as the HTML file, then the CODEBASE attribute may be omitted entirely.
WIDTH=pixels HEIGHT=pixels	These attributes specify the width and height of the space on the HTML page that will be reserved for the applet.
NAME=applet_instance_name (Optional)	A name for the applet may optionally be specified so that other applets on the page may refer to this applet. This facilitates inter-applet communication.
ALIGN=alignment (Optional)	This optional attribute specifies where on the page the applet will appear. Possible values for alignment are: TOP, BOTTOM, LEFT, RIGHT, MIDDLE, ABSMIDDLE, ABSBOTTOM, TEXTTOP, and BASELINE.

(Contd)

Table 14.1 *(Contd)*

`HSPACE=pixels` (Optional)	Used only when ALIGN is set to LEFT or RIGHT, this attribute specifies the amount of horizontal blank space the browser should leave surrounding the applet.
`VSPACE=pixels` (Optional)	Used only when some vertical alignment is specified with the ALIGN attribute (TOP, BOTTOM, etc.,) VSPACE specifies the amount of vertical blank space the browser should leave surrounding the applet.
`ALT=alternate_text` (Optional)	Non-Java browsers will display this text where the applet would normally go. This attribute is optional.

We summarise below the list of things to be done for adding an applet to a HTML document:

1. Insert an <APPLET> tag at an appropriate place in the Web page.
2. Specify the name of the applet's **.class** file.
3. If the **.class** file is not in the current directory, use the codebase parameter to specify
   - the relative path if file is on the local system, or
   - the Uniform Resource Locator (URL) of the directory containing the file if it is on a remote computer.
4. Specify the space required for display of the applet in terms of width and height in pixels.
5. Add any user-defined parameters using <PARAM> tags.
6. Add alternate HTML text to be displayed when a non-Java browser is used.
7. Close the applet declaration with the </APPLET> tag.

14.12 PASSING PARAMETERS TO APPLETS

We can supply user-defined parameters to an applet using <PARAM...> tags. Each <PARAM...> tag has a ***name*** attribute such as ***color,*** and a ***value*** attribute such as ***red***. Inside the applet code, the applet can refer to that parameter by name to find its value. For example, we can change the colour of the text displayed to red by an applet by using a <PARAM...> tag as follows:

```
<APPLET>
<PARAM = color VALUE = "red">
</APPLET>
```

Similarly, we can change the text to be displayed by an applet by supplying new text to the applet through a <PARAM...> tag as shown below:

```
<PARAM NAME = text VALUE = "I love Java">
```

Passing parameters to an applet code using <PARAM> tag is something similar to passing parameters to the **main( )** method using command line arguments. To set up and handle parameters, we need to do two things:

1. Include appropriate <PARAM...> tags in the HTML document.
2. Provide Code in the applet to parse these parameters.

Parameters are passed on an applet when it is loaded. We can define the **init( )** method in the applet to get hold of the parameters defined in the <PARAM> tags. This is done using the **getParameter( )** method, which takes one string argument representing the **name** of the parameter and returns a string containing the value of that parameter.

Program 14.2 shows another version of **HelloJava** applet. Compile it so that we have a class file ready.

Program 14.2 *Applet HelloJavaParam*

```
import java.awt.*;
import   java. applet.*;
public class HelloJavaParam extends Applet
{
     String   str;
     public void init(   )
     {
          str = getParameter( "string" ) ;          // Receiving parameter value
          if (str == null)
               str = "Java";
          str = "Hello" + str;                      // Using the value
     }
     public void paint (Graphics g)
     {
          g.drawString(str, 10, 100).
     }
}
```

Now, let us create HTML file that contains this applet. Program 14.3 shows a Web page that passes a parameter whose NAME is "string" and whose VALUE is "APPLET!" to the applet **HelloJavaParam.**

Program 14.3 *The HTML file for HelloJavaParam applet*

```
<HTML>
     <!Parameterized HTML file>
     <HEAD>
     <TITLE> Welcome to Java Applets </TITLE>
     <HEAD>
     <BODY>
          <APPLET CODE = HelIoJavaParam.class
               WIDTH = 400
               HEIGHT = 200>
               <PARAM NAME = "string"
               VALUE = "Applet!">
          </APPLET>
     </BODY>
</HTML>
```

Save this file as **HelloJavaParam.html** and then run the applet using the applet viewer as follows:

```
appletviewer HelloJavaParam.html
```

This will produce the result as shown in Fig. 14.8.

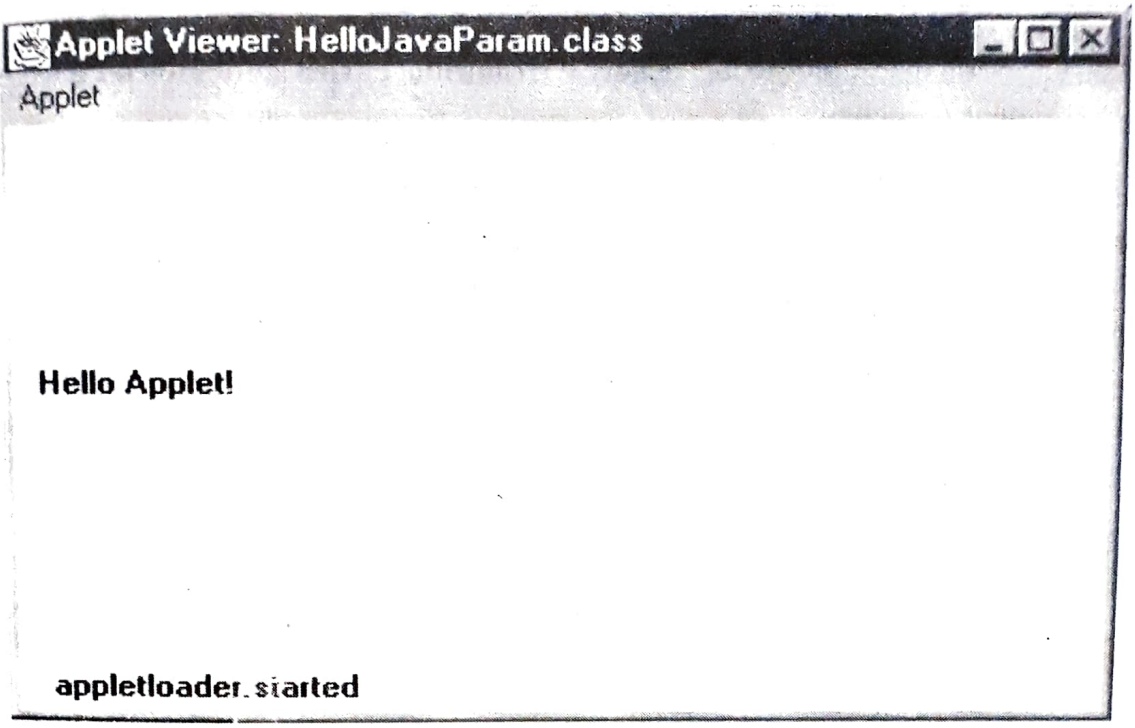


Fig. 14.8 *Displays of HelloJavaParam applet*

Now, remove the <PARAM> tag from the HTML file and then run the applet again. The result will be as shown in Fig. 14.9.

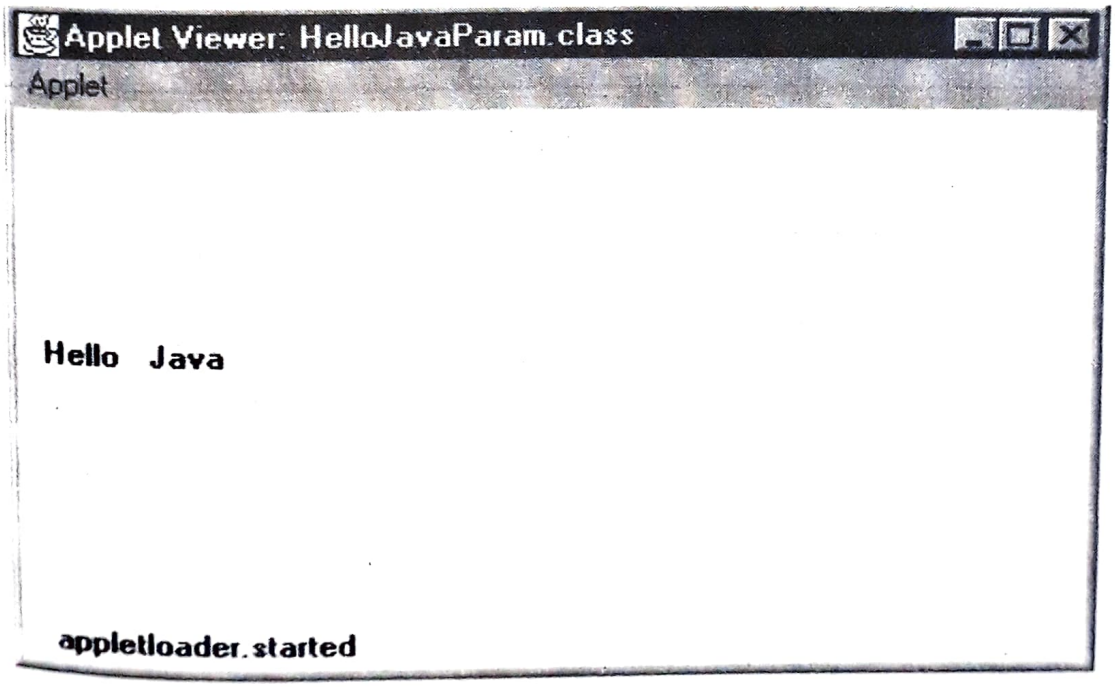


Fig. 14.9 *Output without PARAM tag*

14.13 ALIGNING THE DISPLAY

We can align the output of the applet using the ALIGN attribute. This attribute can have one of the nine values:

LEFT, RIGHT, TOP, TEXT TOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM.

For example, ALGN = LEFT will display the output at the left margin of the page. All text that follows the ALIGN in the Web page will be placed to the right of the display. Program 14.4 shows a HTML file for our **HelloJava** applet shown in Program 14.1.

Program 14.4 *HTML file with ALIGN attribute*

```
<HTML>
     <HEAD>
     <TILTE> Here is an applet </TITLE>
     </HEAD>
     <BODY>
          <APPLET CODE = HelloJava.class
               WIDTH  = 400
               HEIGHT = 200
               ALIGN  = RIGHT >
          </APPLET>
     </BODY>
</HTML>
```

The alignment of applet will be seen and appreciated only when we run the applet using a Java-capable browser. Figure 14.10 shows how an applet and text surrounding it might appear in a Java-capable browser. All the text following the applet appears to the left of that applet.

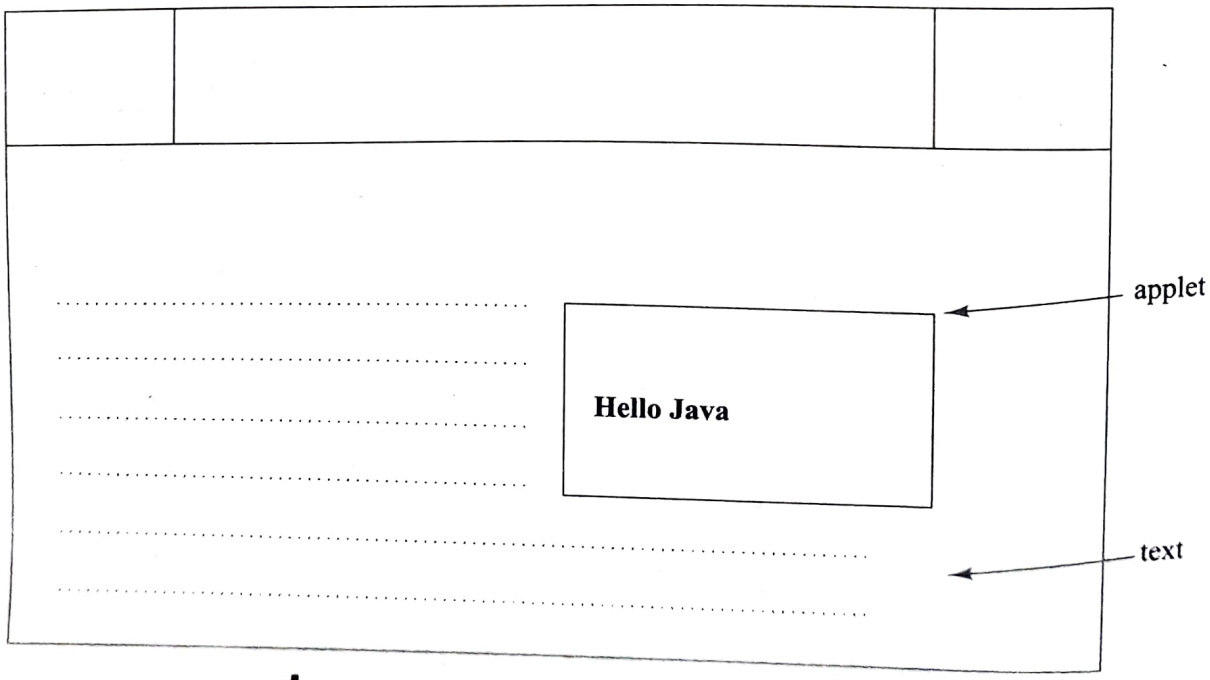


Fig. 14.10 *An applet aligned right*

14.14 MORE ABOUT HTML TAGS

We have seen and used a few HTML tags. HTML supports a large number of tags that can be used to control the style and format of the display of Web pages. Table 14.2 lists important HTML tags and their functions.

Table 14.2 *HTML Tags and Their Functions*

Tag	*Function*
<HTML>....</HTML>	Signifies the beginning and end of a HTML file.
<HEAD>....</HEAD>	This tag may include details about the Web page. Usually contains <TITLE> tag within it.
<TITLE>....</TITLE>	The text contained in it will appear in the title bar of the browser.
<BODY>....</BODY>	This tag contains the main text of the Web page. It is the place where the <APPLET> tag is declared.
<H1>....</H1> <H6>....<H/6>	Header tags. Used to display headings. <H1> creates the largest font header, while <H6> creates the smallest one.
<CENTER> ....<CENTER>	Places the text contained in it at the centre of the page.
<APPLET ...>	<APPLET ...> tag declares the applet details as its attributes.
<APPLET>....</APPLET)	May hold optionally user-defined parameters using <PARAM> Tags.
<PARAM....>	Supplies user-defined parameters. The <PARAM> tag needs to be placed between the <APPLET> and </APPLET> tags. We can use as many different <PARAM> tags as we wish for each applet.
<B>....<B>	Text between these tags will be displayed in bold type.
 	Line break tag. This will skip a line. Does not have an end tag.
<P>	Para tag. This tag moves us to the next line and starts a paragraph of text. No end tag is necessary.
<IMG...>	This tag declares attributes of an image to be displayed.
<HR>	Draws a horizontal rule.
<A....> </A>	Anchor tag used to add hyperlinks.
<FONT ...> ... </FONT>	We can change the colour and size of the text that lies in between <FONT> and </FONT> tags using COLOR and SIZE attributes in thetag<FONT...>.
<! . . . .>	Any text starting with a < ! mark and ending with a > mark is ignored by the Web browser. We may add comments here. A comment tag may be placed anywhere in a Web page.

14.15 DISPLAYING NUMERICAL VALUES

In applets, we can display numerical values by first converting them into strings and then using the **drawString( )** method of **Graphics** class. We can do this easily by calling the **ValueOf( )** method of **String** class. Program 14.5 illustrates how an applet handles numerical values.

Program 14.5 *Displaying numerical values*

```
import java.awt.*;
import java.applet.*;
public class NumValues extends Applet
{
     public void paint (Graphics g)
     {
          int value1 = 10;
          int value2 = 20;
          int sum = value1 + value2;
          String s = "sum:" + String.valueOf(sum);
          g.drawString(s, 100, 100);
     }
}
```

The applet Program 14.5 when run using the following HTML file displays the output as shown in Fig. 14.11.

```
<html>
<applet
     code = NumValues.class
     width = 300
     height = 300 >
</applet>
</html>
```

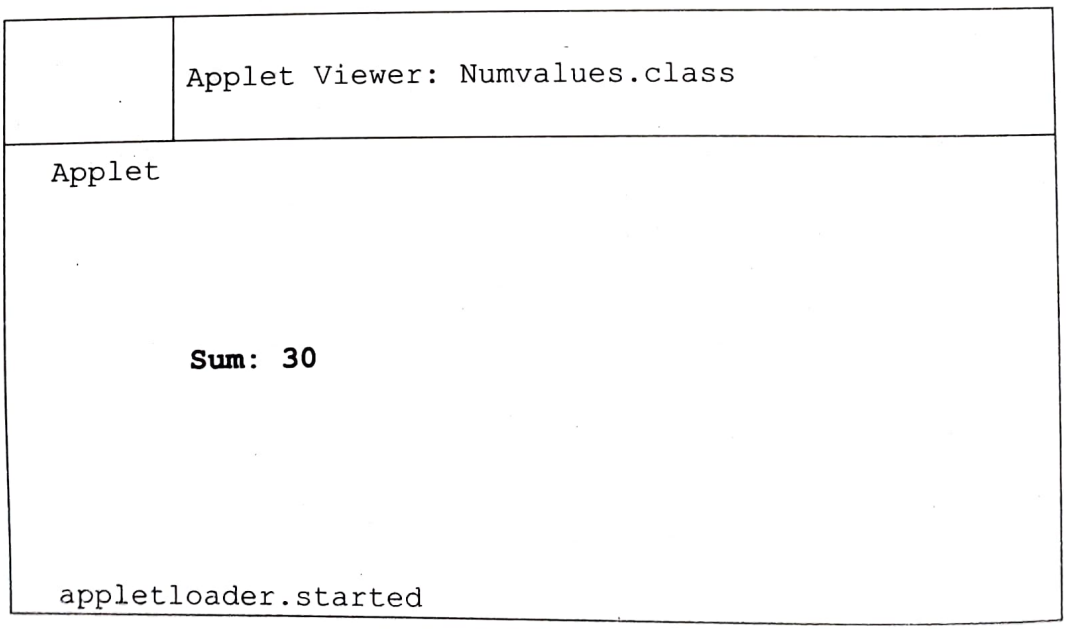


Fig. 14.11 *Output of Program 14.5*

14.16 GETTING INPUT FROM THE USER

Applets work in a graphical environment. Therefore, applets treat inputs as text strings. We must first create an area of the screen in which user can type and edit input items (which may be any data type). We can do

this by using the **TextField** class of the applet package. Once text fields are created for receiving input, we can type the values in the fields and edit them, if necessary.

Next step is to retrieve the items from the fields for display of calculations, if any. Remember, the text fields contain items in string form. They need to be converted to the right form, before they are used in any computations. The results are then converted back to strings for display. Program 14.6 demonstrates how these steps are implemented.

Program 14.6 *Interactive input to an applet*

```
import java.awt.*;
import java.applet.*;
public class UserIn extends Applet
{
   TextField text1, text2;
   public void init( )
   {
      text1 = new TextField(8);
      text2 = new TextField(8);
      add (text1);
      add (text2);
      text1.setText ("0");
      text2.setText ("0");
   }
   public void paint (Graphics g)
   {
      int x = 0, y = 0, z = 0;
      String s1, s2, s;
      g.drawString "Input a number in each box", 10, 50);
      try
      {
            s1 = text1.getText( );
            x = Integer.parseInt(s1);
            s2 = text2.getText( );
            y = Integer.parseInt(s2);
      }
      catch (Exception ex) { }
      z = x + y;
      s = String.val ueOf (z);
      g.drawString ("THE SUM IS:", 10, 75);
      g.drawString (s, 100, 75);
   }
   public Boolean action (Event event, Object object)
   {
      repaint ( );
      return true;
   }
}
```

Run the applet **UserIn** using the following steps:

1. Type and save the program (.java file)
2. Compile the applet (.class file)
3. Write a HTML document (.html file)

```
<html>
<applet
```

```
        code = UserIn.class
        width = 300
        height = 200 >
</applet>
</html>
```

4. Use the **appletviewer** to display the results

When the applet is up and running, enter a number into each text field box displayed in the applet area, and then press the Return Key. Now, the applet computes the sum of these two numbers and displays the result as shown in Fig. 14.12.

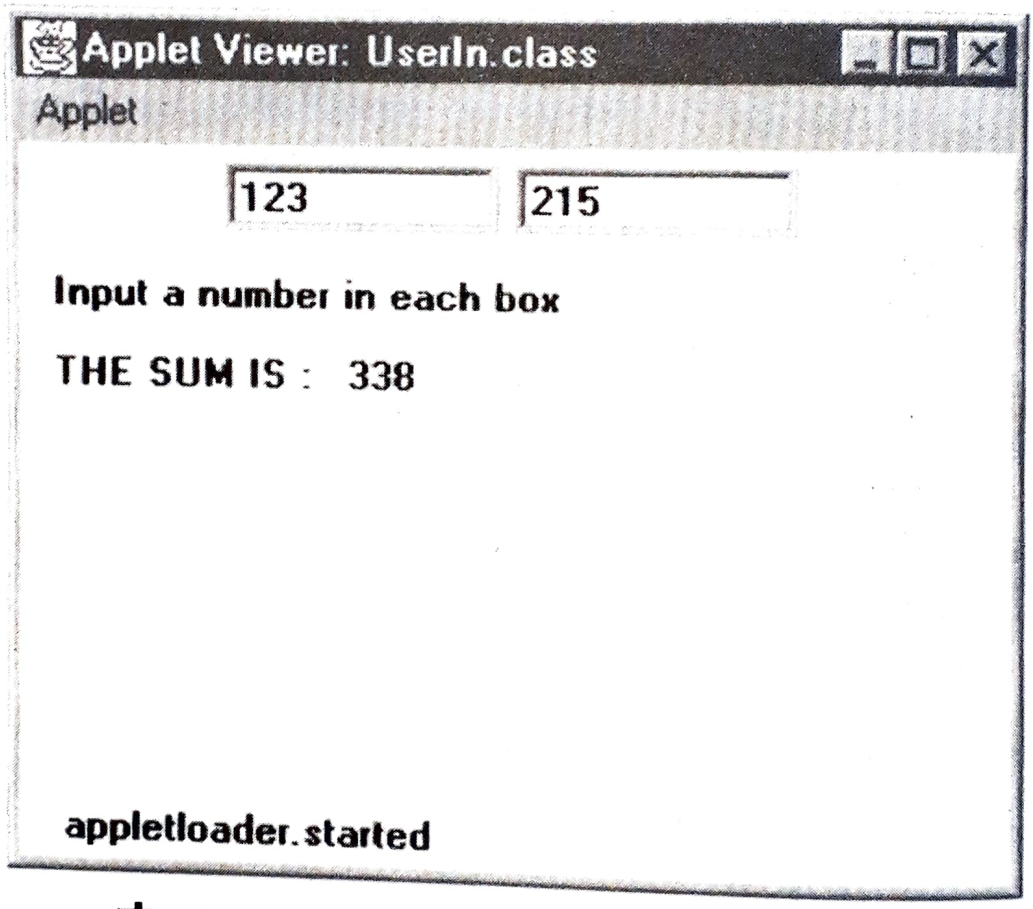


Fig. 14.12 *Interactive computing with applets*

Program Analysis

The applet declares two **TextField** objects at the beginning.

```
TextField text1, text2;
```

These two objects represent text boxes where we type the numbers to be added.

Next, we override the **init( )** method to do the following:

1. To create two text field objects to hold strings (of eight character length).
2. To add the objects to the applet's display area.
3. To initialize the contents of the objects to zero.

Then comes the **paint( )** method where all the actions take place. First, three integer variables are declared, followed by three string variables. Remember, the numbers entered in the text boxes are in string form and therefore they are retrieved as strings using the **getText( )** method and then they are converted to numerical values using the **parseInt( )** method of the **Integer** class.

After retrieving and converting both the strings to integer numbers, the **paint( )** method sums them up and stores the result in the variable **z**. We must convert the numerical value in **z** to a string before we attempt to display the answer. This is done using the **ValueOf( )** method of the **String** class.

14.17 EVENT HANDLING

As the name suggests, event handling is a mechanism that is used to handle events generated by applets. An event could be the occurrence of any activity such as a mouse click or a key press. In Java, events are regarded as method calls with a certain task performed against the occurrence of each event.

Some of the key events in Java are:

- **ActionEvent** is triggered whenever a user interface element is activated, such as selection of a menu item.
- **ItemEvent** is triggered at the selection or deselection of an itemized or list element, such as check box.
- **TextEvent** is triggered when a text field is modified.
- **WindowEvent** is triggered whenever a window-related operation is performed, such as closing or activating a window.
- **KeyEvent** is triggered whenever a key is pressed on the keyboard.

Each event is associated with an event source and an event handler. Event source refers to the object that is responsible for generating an event whereas event listener is the object that is referred by event source at the time of occurrence of an event.

Event Sources

It is not necessary that a source generates only one type of event; it can generate events of multiple types. However, it is mandatory that the source object registers listener objects corresponding to each of its event types. The registration of a listener object with an event ensures that on occurrence of the event, the corresponding listener object is notified for taking appropriate action. Following is the syntax for registering a listener for an event:

```
public void add<Type>Listener(<Type>Listener EveList)
```

Here, **Type** is the name of the event and **EveList** is the corresponding listener object reference.

Event Listeners

The **event listener** object contains methods for receiving and processing event notifications sent by the source object. These methods are implemented from the corresponding listener interface contained in the **java.awt.event** package.

Event Classes

All the events in Java have corresponding event classes associated with them. Each of these classes is derived from one single super class, i.e., **EventObject**. It is contained in the **java.util package**. The **EventObject** class contains the following two important methods for handling events:

- **getSource()**: Returns the event source.
- **toString()**: Returns a string containing information about the event source.

The immediate subclass of **EventObject** is the **AWTEvent** class from which all the AWT-based event classes are derived. Figure 14.13 shows hierarchy of event-based classes:

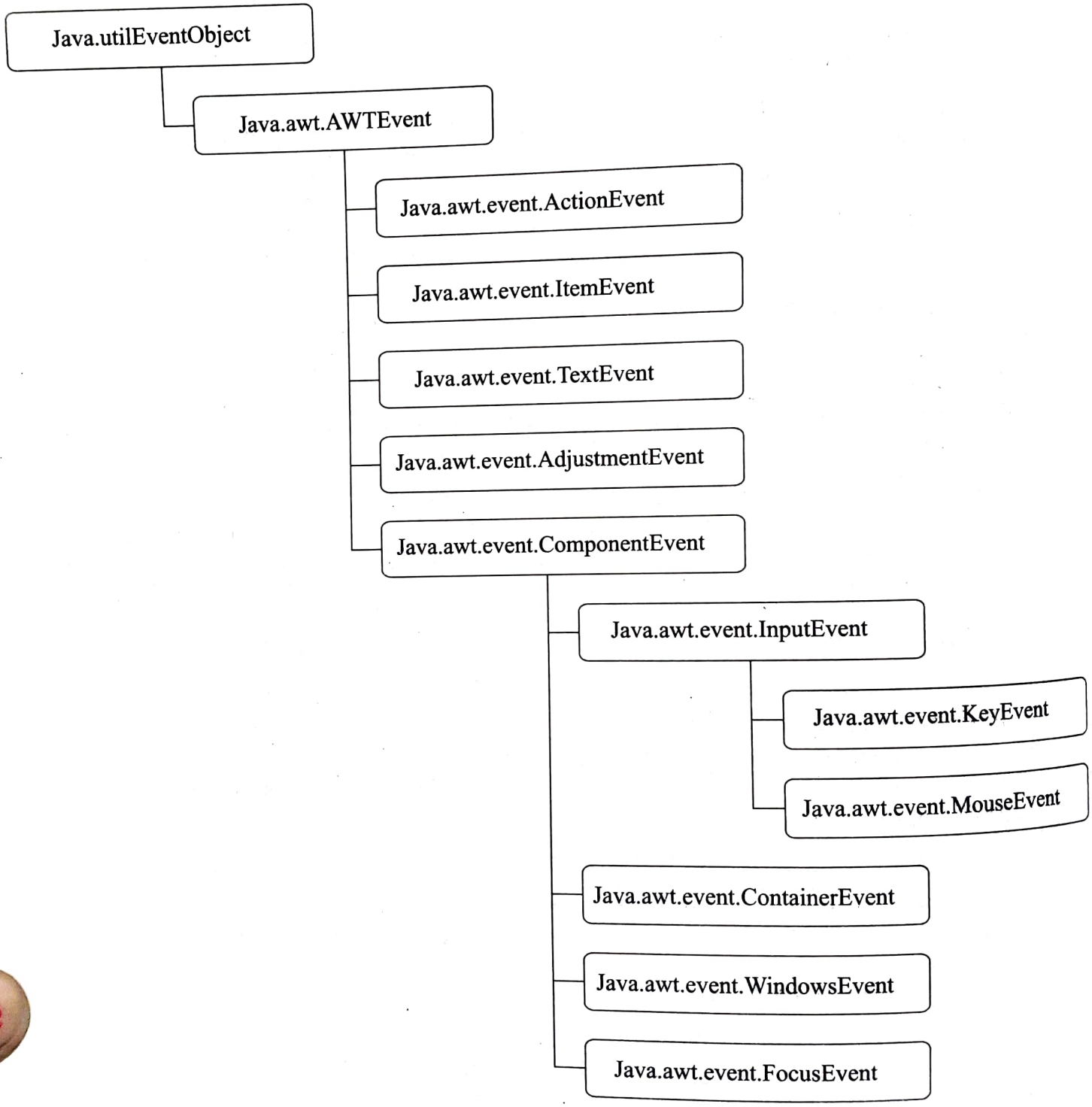


Fig. 14.13 *Hierarchy of event-based classes in Java*

The following is a sample program depicting the usage of keyboard event in an applet.

Program 14.7 *Keyboard event in an applet*

```
<html>
<body>
```

```
<applet code=eg_event.class width =200 height=200>
</applet>
</body>
</html>

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

public class eg_event extends Applet implements KeyListener
{
public void init()
{
addKeyListener(this);
}
public void keyTyped(KeyEvent KB){}
public void keyReleased(KeyEvent KB)
{
showStatus("Key on the keyboard is released");
}
public void keyPressed(KeyEvent KB){
showStatus("A key on the keyboard is pressed");
}
Font f1 = new Font("Courier New",Font.BOLD, 20);
public void paint(Graphics GA){
GA.setFont(f1);
GA.setColor(Color.blue);
GA.drawString("This applet sense the up/down motion of keys",20,120);
}}
```

Output of Program 14.7:

Applet

This applet sense the up/down motion of keys

Key on the keyboard is released

This program detects the key-press activities from the keyboard and flashes the corresponding message accordingly.

14.18 SUMMARY

Applets are Java programs developed for use on the Internet. They provide a means to distribute interesting, dynamic, and interactive applications over the World Wide Web. We have learned the following about applets in this chapter:

How do applets differ from applications,
How to design applets,
How to design a Web page using HTML tags,
How to execute applets, and
How to provide interactive input to applets.

KEY TERMS

Applet, Local applet, Remote applet, Web page, HTML tag, APPLET tag, Applet life cycle.

REVIEW QUESTIONS

14.1 What is an applet?
14.2 What is a local applet?
14.3 What is a remote applet?
14.4 Explain the client/server relationship as applied to Java applets.
14.5 How do applets differ from application programs?
14.6 Discuss the steps involved in developing and running a local applet.
14.7 Discuss the steps involved in loading and running a remote applet.
14.8 Describe the various sections of Web page.
14.9 How many arguments can be passed to an applet using <PARAM> tags?
14.10 Why do applet classes need to be declared as public?
14.11 Describe the different stages in the life cycle of an applet. Distinguish between **init( )** and **start( )** methods.
14.12 Develop an applet that receives three numeric values as input from the user and then displays the largest of the three on the screen. Write a HTML page and test the applet.

DEBUGGING EXERCISES

14.1 Find errors in the following code for drawing set of nested Rectangles.

```
importjava.awt.*;
importjava.applet.Applet;
public class Rectangles extends Applet
{
        public void paint(Graphics g)
        {
            int inset;
            int rectWidth, rectHeight;
            g.SetColor(Color.blue);
            g.fillRect(0,0,300,160);
            inset = 0;
            rectWidth = 299;
            rectHeight= 159;
```

```
            g.setColor(Color.red);
            g.drawStr("Rectangles",150,200);
            while (rectWidth> = 0 && rectHeight> = 0)
            {
                g.drawRect(inset, inset, rectWidth, rectHeight)
            inset += 15;
            rectWidth -= 30;
            rectHeight -= 30;
        }
    }
}
```

14.2 The following code converts temperature values. Will the code display the new value on moving the scrollbar?

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class CelsiusValue extends Applet implements AdjustmentListener
{
    private Scrollbar bar;
    private int old, newtemp = 0;
    private inft fahr = 32;
    public void init()
    {
    bar = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 100);
    bar.addAdjustmentListener(this);
    setLayout(new BorderLayout( ) );
    add("North", bar);
}
public void paint(Graphics g)
{
    g.drawSTring("Celsius = " + newtemp, 30, 50);
    g.drawString ("Fahrenheit = " + fahr,  30,  70);
}
public void adjustmentValueChanged(AdjustmentEvent e)
{
    newtemp = bar.getValue( );
    if (newtemp ! = old)
    {
        fahr = newtemp * 9 / 5 + 32;
        old = newtemp;
    }
}
}
```

14.3 Given code creates an expanding ring on mouse click event. Does the code show the desired output?

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class MouseRing extends Applet implements MouseListener
{
    private int   x = 100, y = 100;
    private int   pauseLength;
    public void   init( )
    {
        pauseLength = Integer.parseInt(getParameter("PauseLength"));
```

```
        setBackground(Color.white);
    }
    public void paint(Graphics g)
    {
    int count = 0;
    while (count < 100)
        {
            int radius = 5*count;
            int diameter = 2*radius;
            g.setColor(Color.black);
            g.drawOval(-radius, y-radius,  diameter,  diameter);
            // Draw pause(pauseLength);
            g.setColor(Color.white);
            g.drawOval(x-radius, y-radius, diameter, diameter)
            // Erase!
            count = count+1;
        }
    }
        private void pause(int howLong)
        {
            for (int count = 0; count < howLong; count++);
        }
        public void mouseClicked(MouseEvent e)
        {
            x = e.getX();
            y = e.getY();
            repaint();
        }
        public void mouseExited(MouseEvent e) { }
        public void mouseEntered(MouseEvent e) { }
        public void mousePressed(MouseEvent e) { }
        public void mouseReleased(MouseEvent e) { }
    }
```

.4 Using parameter, an applet provides answers to different questions. Correct the code.

Question.html

```
<html> <head>
<title>Questions and Answers</title>
</hnead>
<body>
<APPLET CODE=Question.class
        WIDTH=400 HEIGHT = 100>
    <PARAM NAME=question VALUE="What is Inheritance?">
    <PARAM NAME=answer VALUE="Getting the properties of one class into
    another">
</APPLET>
</body>
</html>
```

Question.java

```
import java.awt.*;
import java.awt.event.*;
import java.appi et.Applet;
public class Question extends Applet implements ActionListener
{
    String theQuestion;
    String theAnswer = " ";
```

```
Button reveal = new Button("Click to know the answer");
public void init()
{
theQuestion = getParameter("ques");
add(reveal);
reveal.addActionListener(this);
}
public void paint(Graphics g)
{
     g.setColor(Color.black);
     g.drawString(theQuestion, 10, 50);
     g.setColor(Color.red);
     g.drawString(theAnswer, 10, 70);
}
public void actionPerformed(ActionEvent e)
{
     theAnswer = getParameter("answer");
     repaint();
}
}
```

14.5 Given applet shows the sequence of events called in an applet. Will the message defined in destroy() event be shown?

```
import java.awt.*;
importjava.applet.Applet;
public classAllMethodsApplet extends Applet
    {
        TextrArea messages = new TextArea(8, 30);
        public AllMethodsApplet()
        {
             messages.append("Constructor called/n");
        }
        public void init()
        {
             add(messages);
             messages.append ("Init called/n");
     }
     public void start()
     {
             messages.append ("Start called/n" );
     }
     public void stop( )
     {
             messages.append ("Stop called/n");
     }
     public void destory( )
     {
          messages.append ("Destroy called/n");
     }
     public void paint(Graphics display)
     {
          messages.append ("Paint called/n");
          Dimension size = getSize( );
          display.drawRect( 0, 0, size.width-1, size.height-1);
     }
}
```

CHAPTER 15

Graphics Programming

15.1 INTRODUCTION

One of the most important features of Java is its ability to draw graphics. We can write Java applets that draw lines, figures of different shapes, images, and text in different fonts and styles. We can also incorporate different colours in display.

Every applet has its own area of the screen known as *canvas,* where it creates its display. The size of an applet's space is decided by the attributes of the <APPLET...> tag. A Java applet draws graphical image inside its space using the coordinate system as shown in Fig. 15.1.

Java's coordinate system has the origin (0, 0) in the upper-left corner. Positive x values are to the right, and positive y values are to the bottom. The values of coordinates x and y are in pixels.

15.2 THE GRAPHICS CLASS

Java's **Graphics** class includes methods for drawing many different types of shapes, from simple lines to polygons to text in a variety of fonts. We have already seen how to display text using the **point( )** method and a **Graphics** object.

To draw a shape on the screen, we may call one of the methods available in the **Graphics** class. Table 15.1 shows the most commonly used drawing methods in the **Graphics** class. All the drawing methods have arguments representing end points, corners, or starting locations of a shape as values in the applet's coordinate system. To draw a shape, we only need to use the appropriate method with the required arguments.

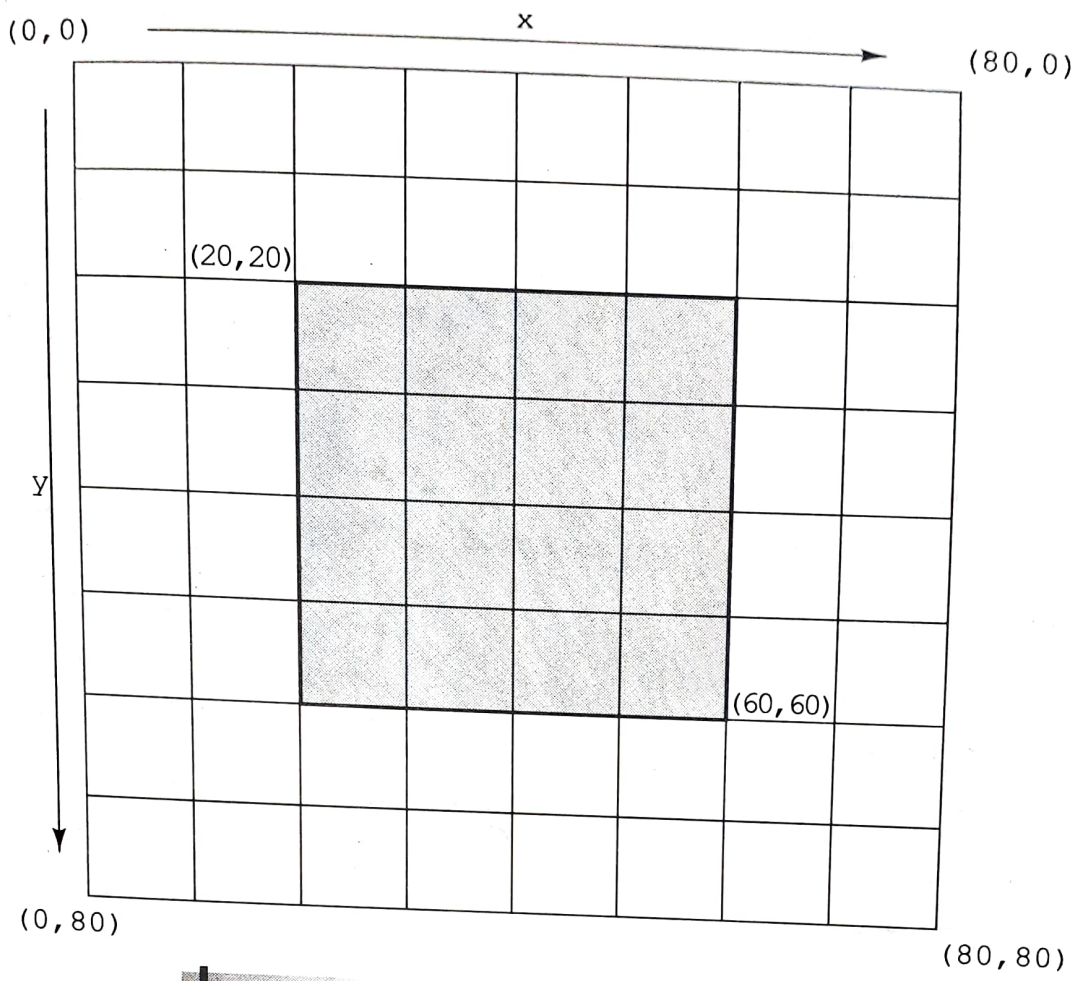


Fig. 15.1 *Coordinate system of Java*

Table 15.1 *Drawing Methods of the Graphics Class*

Method	*Description*
`clearRect ( )`	Erases a rectangular area of the canvas.
`copyArea ( )`	Copies a rectangular area of the canvas to another area.
`drawArc ( )`	Draws a hollow arc.
`drawLine ( )`	Draws a straight line.
`drawOval ( )`	Draws a hollow oval.
`drawPolygon ( )`	Draws a hollow polygon.
`drawRect ( )`	Draws a hollow rectangle.
`drawRoundRect ( )`	Draws a hollow rectangle with rounded corners.
`drawstring ( )`	Displays a text string.
`fillArc ( )`	Draws a filled arc.
`fillOval ( )`	Draws a filled oval.
`fillPolygon ( )`	Draws a filled polygon.
`fillRect ( )`	Draws a filled rectangle
`fillRoundRect ( )`	Ddraws a filled rectangle with rounded corners.
`getColor ( )`	Retrieves the current drawing colour.
`getFont ( )`	Retrieves the currently used font.
`getFontMetrics ( )`	Retrieves information about the current font.
`setColor ( )`	Sets the drawing colour.
`setFont ( )`	Sets the font.

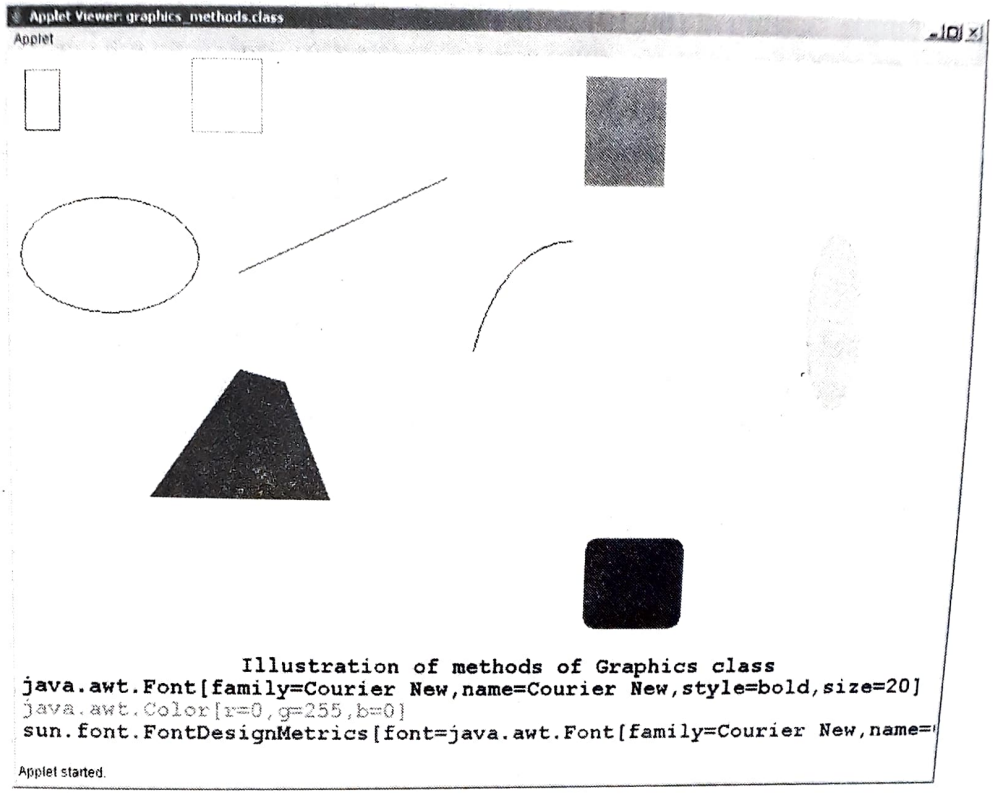
Program 15.1 is a quick demonstration of the above-mentioned drawing methods of the Graphics class.

Program 15.1 *Using methods of Graphics class*

```
<html>
<body>
<applet code=graphics_methods.class width =200 height=200>
</applet>
</body>
</html>

import java.awt.*;
import java.applet.*;
public class graphics_methods extends Applet
{
String s = new String( );
String s1 = new String( );
String s2 = new String( );
Font f1 = new Font("Courier New",Font.BOLD, 20);
public void paint(Graphics GA)
{
GA.setFont(f1);
GA.setColor(Color.blue);
GA.drawString("Illustration of methods of Graphics class",200,520);
Font f2 = GA.getFont( );
s = f2.toString();
GA.drawString(s,5,540);
GA.setColor(Color.green);
Color col = GA.getColor( );
s2 = col.toString();
GA.drawString(s2,5,560);
GA.fillRect(500, 15, 70, 90);
GA.drawRect(160, 5, 60, 60);
GA.drawOval(10,120,155,95);
GA.setColor(Color.yellow);
GA.fillOval(700,140,50,150);
GA.setColor(Color.black);
GA.drawLine(380,100, 200, 180);
GA.drawArc(400,150,180,280,90,70);
int x2[] = {200,120,280,240};
int z2=4,y2[ ] = {260,370,370,270} ;
GA.setColor(Color.blue);
GA.fillPolygon(x2, y2, z2);
GA.setColor(Color.red);
GA.drawRect(15, 15, 30, 50);
FontMetrics f3 = GA.getFontMetrics( );
s1 = f3.toString( );
GA.drawString(s1,5,580);
GA.setColor(Color.magenta);
GA.fillRoundRect(510,400,90, 80,20,20 );}
}
```

Output of Program 15.1:



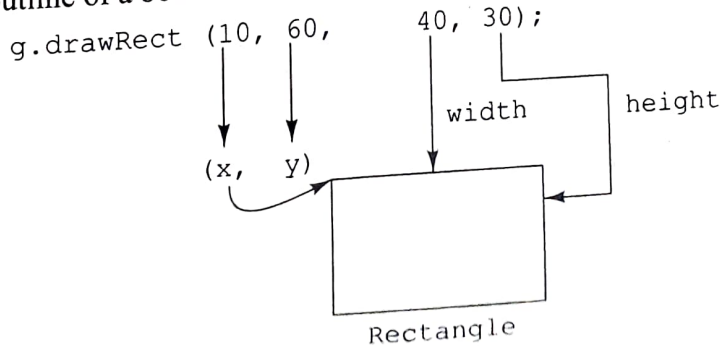
15.3 LINES AND RECTANGLES

The simplest shape we can draw with the **Graphics** class is a line. The **drawLine** ( ) method takes two pair of coordinates, (xl, yl) and (x2, y2) as arguments and draws a line between them. For example, the following statement draws a straight line from the coordinate point (10,10) to (50, 50):

```
g.drawLine (10,10, 50,50) ;
```

The **g** is the **Graphics** object passed to **paint( )** method.

Similarly, we can draw a rectangle using the **drawRect( )** method. This method takes four arguments. The first two represent the x and y coordinates of the top left corner of the rectangle, and the remaining two represent the width and the height of the rectangle. For example, the statement will draw a rectangle starting at (10,60) having a width of 40 pixels and a height of 30 pixels. Remember that the **drawRect** ( ) method draws only the outline of a box.



We can draw a solid box by using the method **fillRect** ( ). This also takes four parameters (as drawRect) corresponding to the starting point, the width and the height of the rectangle. For example, the statement

```
g.fillRect (60, 10, 30, 80) ;
```

will draw a solid rectangle starting at (60,10) with a width of 30 pixels and a height of 80 pixels.

We can also draw rounded rectangles (which are rectangles with rounded edges), using the methods **drawRoundRect ( )** and **fillRoundRect ( )**. These two methods are similar to **drawRect ( )** and **fillRect ( )** except that they take two extra arguments representing the width and height of the angle of corners. These extra parameters indicate how much of corners will be rounded. Example:

```
g.drawRoundRect (10, 100, 80, 50, 10, 10) ;
g.fillRoundRect (20, 110, 60, 30, 5, 5) ;
```

Program 15.2 is an applet code that draws three lines, a rectangle, a filled rectangle, a rounded rectangle and a filled rounded rectangle. Note that the filled rounded one is drawn inside the rounded rectangle. The output of the applet **LineRect** under **appletviewer** is shown in Fig. 15.2.

Program 15.2 *Drawing lines and rectangles*

```
import java.awt.*;
import java.applet.*;
public class LineRect extends Applet
{
     public void paint (Graphics g)
     {
          g.drawLine (10, 10, 50, 50) ;
          g.drawRect (10, 60, 40, 30) ;
          g.fillRect (60, 10, 30, 80) ;
          g.drawRoundRect (10, 100, 80, 50, 10, 10) ;
          g.fillRoundRect (20, 110, 60, 30, 5, 5) ;
          g.drawLine (100, 10, 230, 140) ;
          g.drawLine (100, 140, 230, 10) ;
     }
}
<APPLET
     CODE = LineRect.class
     WIDTH = 250
     HEIGHT = 200>
</APPLET>
```

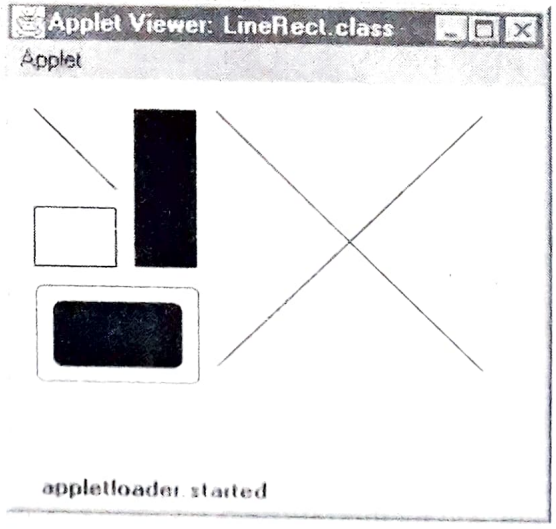


Fig. 15.2 *Output of LineRect applet*

15.4 CIRCLES AND ELLIPSES

The **Graphics** class does not have any method for circles or ellipses. However, the **drawOval( )** method can be used to draw a circle or an ellipse. Ovals are just like rectangles with overly rounded corners as shown in Fig. 15.3. Note that the figure is surrounded by a rectangle that just touches the edges. The **drawOval( )** method takes four arguments: the first two represent the top left corner of the imaginary rectangle and the other two represent the width and height of the oval itself. Note that if the width and height are the same, the oval becomes a circle. The oval's coordinates are actually the coordinates of an enclosing rectangle.

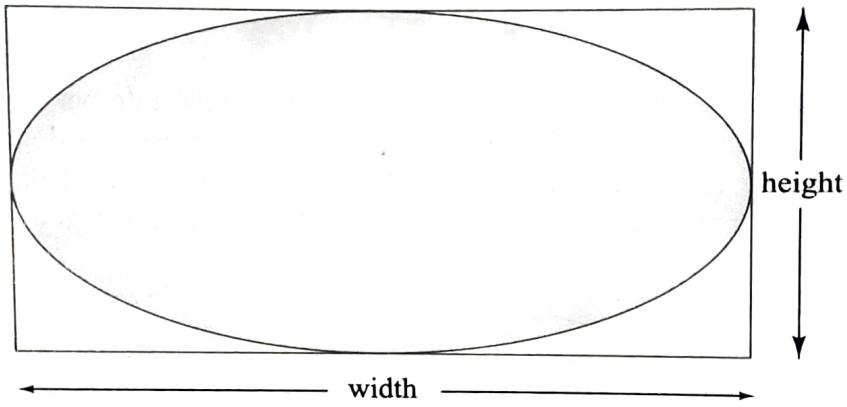


Fig. 15.3 *Oval within an imaginary rectangle*

Like rectangle methods, the **drawOval( )** method draws outline of an oval, and the **fillOval( )** method draws a solid oval. The code segment shown below draws a filled circle within an oval (see Fig. 15.4).

```
public void paint (Graphics g)
{
     g.drawOval (20, 20, 200, 120) ;
     g.setColor (Color.green);
     g.fillOval (70, 30, 100, 100) ; // This is a circle.
}
```

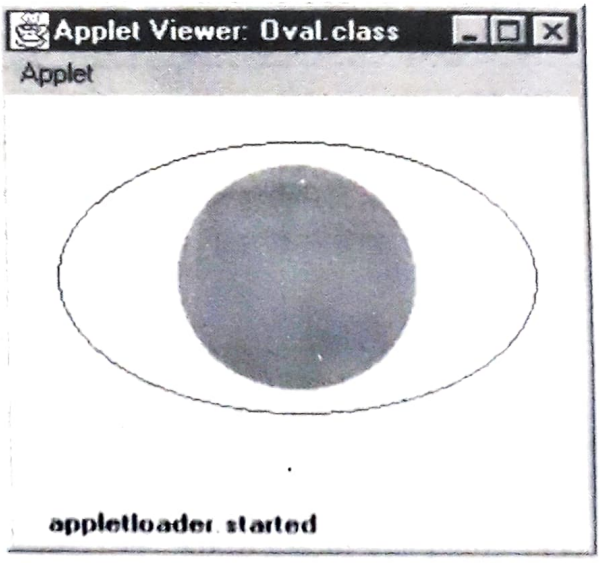


Fig. 15.4 *A filled circle within an ellipse*

We can draw an object using a color object as follows:

```
g.setColor (Color.green) ;
```

After setting the color, all drawing operations will occur in that color.

15.5 DRAWING ARCS

An arc is a part of an oval. In fact, we can think of an oval as a series of arcs that are connected together in an orderly manner. The **drawArc( )** designed to draw arcs takes six arguments. The first four are the same as the arguments for **drawOval( )** method and the last two represent the starting angle of the arc and the number of degrees (sweep angle) around the arc.

In drawing arcs, Java actually formulates the arc as an oval and then draws only a part of it as dictated by the last two arguments. Java considers the three O'clock position as zero degree position and degrees increase in anti-clockwise direction as shown in Fig. 15.5. So, to draw an arc from 12:00 O'clock position to 6:00 O'clock position, the starting angle would be 90, and the sweep angle would be 180.

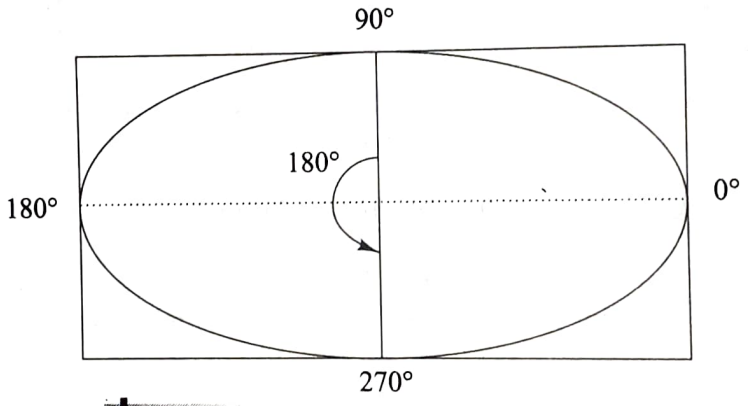


Fig. 15.5 *Arc as a part of an oval*

We can also draw an arc in backward direction by specifying the sweep angle as negative. For example, if the last argument is –135° and the starting angle is 45°, then the arc is drawn as shown in Fig. 15.6.

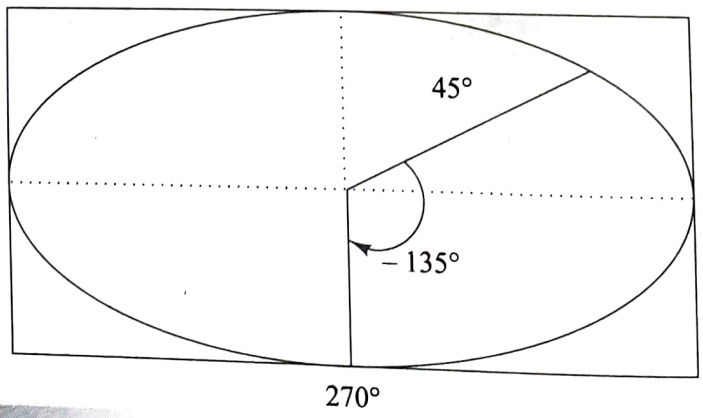


Fig. 15.6 *Drawing an arc in clockwise direction*

We can use the **fillArc( )** method to fill the arc. Filled arcs are drawn as if they were sections of a pie. Instead of joining the two end points, they are joined to the centre of the oval. Program 15.3 shows an applet that draws a human face as shown in Fig. 15.7.

Program 15.3 *Applet for drawing a human face*

```
import java.awt.*;
import java.applet.*;
public class Face extends Applet
{
    public void paint (Graphics g)
    {
        g.drawOval (40, 40, 120, 150) ;          // Head
        g.drawOval (57, 75, 30, 20) ;            // Lefteye
        g.drawOval (110, 75, 30, 20) ;           // Right eye
        g.fillOval (68, 81, 10, 10) ;            // Pupil (left)
        g.fillOval (121, 81, 10, 10) ;           // Pupil (right)
        g.drawOval (85, 100, 30, 30) ;           // Nose
        g.fillArc (60, 125, 80, 40, 180, 180) ;  // Mouth
        g.drawOval (25, 92, 15, 30) ;            // Left ear
        g.drawOval (160, 92, 15, 30) ;           // Right ear
    }
}
```

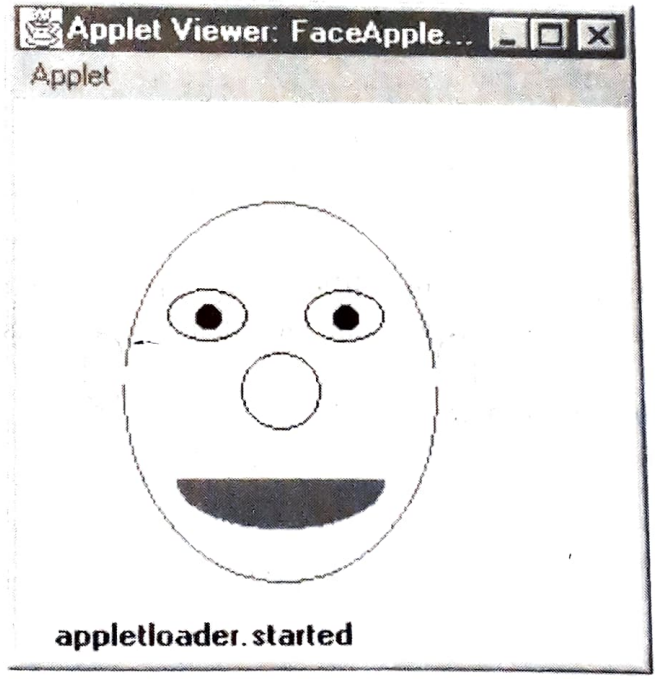


Fig. 15.7 *Output of the Face applet*

15.6 DRAWING POLYGONS

Polygons are shapes with many sides. A polygon may be considered a set of lines connected together. The end of the first line is the beginning of the second line, the end of the second is the beginning of the third, and so on. This suggests that we can draw a polygon with n sides using the **drawLine( )** method n times in succession. For example, the code given below will draw a polygon of three sides. The output of this code is shown in Fig. 15.8.

```
public void paint (Graphics g)
{
     g.drawLine (10, 20, 170, 40) ;
     g.drawLine (170, 40, 80, 140) ;
     g.drawLine (80, 140, 10, 20) ;
}
```

Note that the end point of the third line is the same as the starting point of the polygon.

We can draw polygons more conveniently using the **drawPolygon( )** method of **Graphics** class. This method takes three arguments:

- An array of integers containing x coordinates
- An array of integers containing y coordinates
- An integer for the total number of points

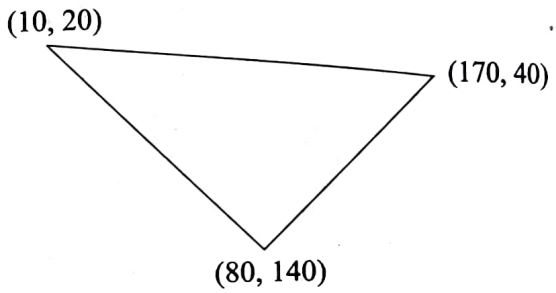


Fig. 15.8 *A polygon with three sides*

It is obvious that x and y arrays should be of the same size and we must repeat the first point at the end of the array for closing the polygon. The polygon shown in Fig. 15.8 can be drawn using the **drawPolygon( )** method as follows:

```
public void paint (Graphics g)
{
     int xPoints [ ] = {10, 170, 80, 10};
     int yPoints [ ] = {20, 40, 140, 20};
     int nPoints [ ] = xPoints.length;
     g.drawPolygon (xPoints, yPoints, nPoints) ;
}
```

We can also draw a filled polygon by using the **fillPolygon( )** method. Program 15.4 illustrates the use of polygon methods to draw both the empty polygons and the filled polygons. Its output is shown in Fig. 15.9.

Program 15.4 *Drawing polygons*

```
import java.awt.*;
import java.applet.*;
public class Poly extends Applet
{
     int x1 [ ] = {20, 120, 220, 20};
     int y1 [ ] = {20, 120, 20, 20};
     int n1 = 4;
     int x2 [ ] = {120, 220, 220, 120};
     int y2 [ ] = {120, 20, 220, 120};
     int n2 = 4;
     public void paint (Graphics g)
     {
          g.drawPolygon (x1, y1, n1) ;
          g.fill Polygon (x2, y2,n2) ;
     }
}
```

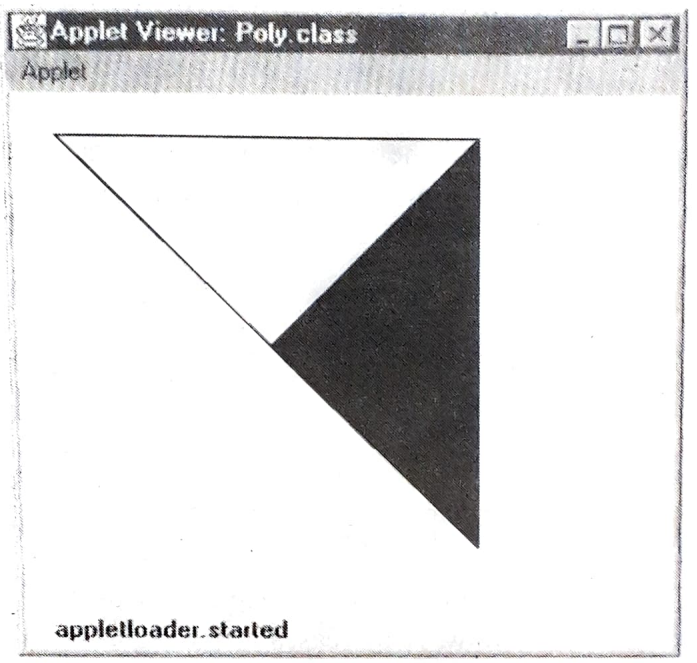


Fig. 15.9 *Output of program 15.4*

Another way of calling the methods **drawPolygon( )** and **fillPolygon( ) is** to use a **Polygon** object. The **Polygon** class enables us to treat the polygon as an object rather than having to deal with individual arrays. This approach involves the following steps:

1. Defining x coordinate values as an array
2. Defining y coordinate values as an array
3. Defining the number of points n
4. Creating a Polygon object and initializing it with the above x, y and n values.
5. Calling the method **drawPolygon( )** or **fillPolygon( )** with the polygon object as arguments

The following code illustrates these steps:

```
public void paint (Graphics g)
{
    int x [ ] = {20, 120, 220, 20};
    int y [ ] = {20, 120, 20, 20};
    int n = x.length;
    Polygon poly = new Polygon (x, y, n) ;
    g.drawPolygon (poly) ;
}
```

The **Polygon** class is useful if we want to add points to the polygon. For example, if we have a polygon object existing, then we can add a point to it as follows:

```
poly.addPoint (x, y) ;
```

The above code may be rewritten using the **addPoint( )** method as follows:

```
public void paint (Graphics g)
{
    Polygon poly = new Polygon ( ) ;
    poly.addPoint (20, 20) ;
    poly.addPoint (120, 120) ;
    poly.addPoint (220, 20) ;
```

```
        poly.addPoint (20, 20) ;
        d.drawPolygon (poly) ;
    }
```

Here, we first create an empty polygon and then add points to it one after another. Finally, we call the **drawPolygon( )** method using the **poly** object as an argument to complete the process of drawing the polygon.

15.7 LINE GRAPHS

We can design applets to draw line graphs to illustrate graphically the relationship between two variables. Consider the table of values shown as follows:

X	0	60	120	180	240	300	360	400
Y	0	120	180	260	340	340	300	180

The variation of Y, when X is increased can be shown graphically using the Program 15.5. The graph is drawn in an area 400 *x* 400 pixels. As we know, X increases to the right and Y increases downwards. This is different from the normal graphical systems that have their origin in the bottom left corner. In order to convert the Java coordinate system to the normal systems, we transform the Y values to N-Y where N is the height of the display area. For example, if the vertical height of the display area is 400 pixels, then the point (5, 5) would be (5, 395) in the new system. The table below shows the new values of X and Y and Program 15.5 gives the applet code that draws a line graph showing the relationship between X and Y. Figure 15.10 shows the graph.

X	0	60	120	180	240	300	360	420
Y	400	280	220	140	60	60	100	220

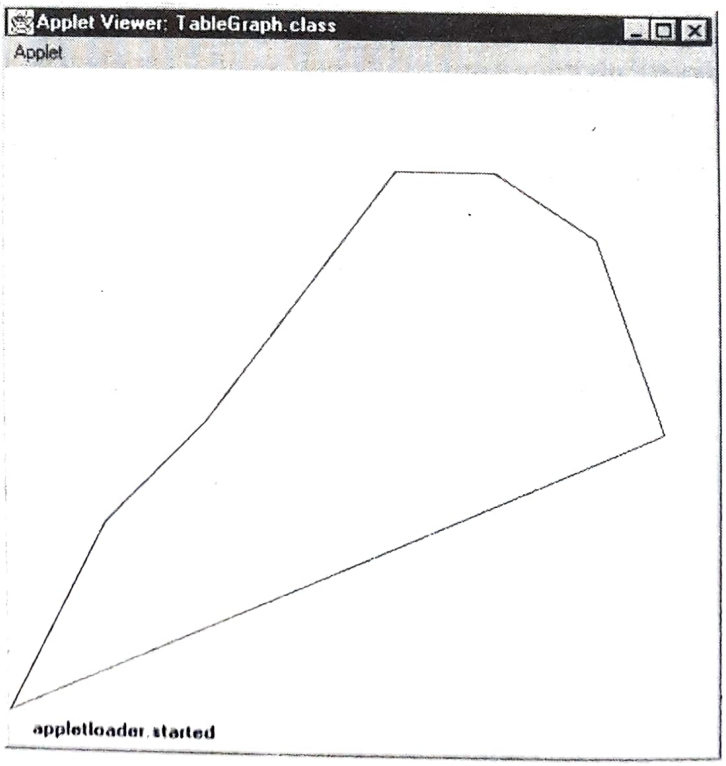


Fig. 15.10 *A line graph*

Program 15.5 *Applet to draw a line graph*

```
import java.awt.*;
import java.applet.*;
public class TableGraph extends Applet
{
     int x [ ] = {0, 60, 120, 180, 240, 300, 360, 400) ;
     int y [ ] = {400, 280, 220, 140, 60, 60, 100, 220);
     int n = x.length ;
     public void paint (Graphics g)
     {
          g.drawPolygon (x, y, n) ;
     }
}
```

15.8 USING CONTROL LOOPS IN APPLETS

We can use all control structures in an applet. Program 15.6 uses a **for** loop for drawing circles repeatedly. When we run the applet shown in Program 15.6, the for loop draws five circles as shown in Fig. 15.11.

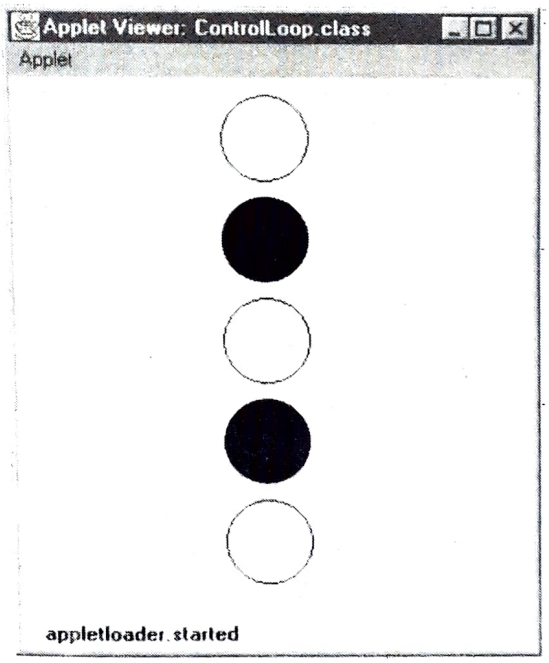


Fig. 15.11 *Output of Program 15.6*

Program 15.6 *Using control loops in applets*

```
import java.awt.*;
import java . applet.*;
public class ControlLoop extends Applet
{
     public void paint (Graphics g)
```

```
    {
        for (int i=0; i<=4; i++)
        {
            if(i%2) == 0)
                g.drawOval (120, i*60+10, 50, 50) ;
        else
            g.fillOval (120, 1*60+10, 50, 50) ;
        }
    }
}
```

15.9 DRAWING BAR CHARTS

Applets can be designed to display bar charts, which are commonly used in comparative analysis of data. The table below shows the annual turnover of a company during the period 1991 to 1994. These values may be placed in a HTML file as PARAM attributes and then used in an applet for displaying a bar chart.

Year	1991	1992	1993	1994
Turnover (Rs Crores)	110	150	100	170

Program 15.7 shows an applet that receives the data values from the HTML page shown in Program 15.8 and displays an appropriate bar chart. The method **getParameter( )** is used to fetch the data values from the HTML file. Note that the method **getParameter( )** returns only string values and therefore we use the wrapper class method **parseInt( )** to convert strings to integer values. Figure 15.12 shows the result of Program 15.7.

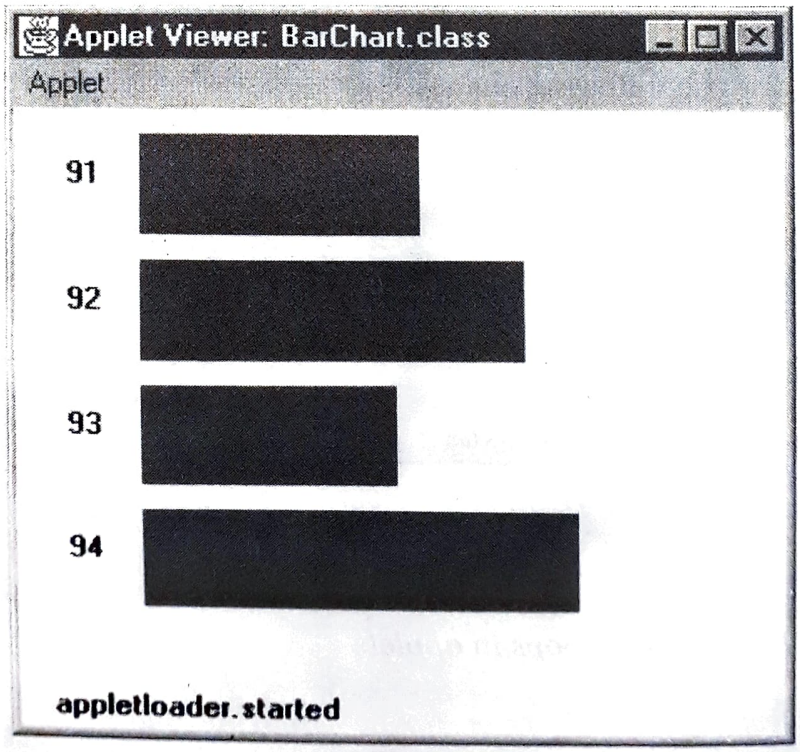


Fig. 15.12 *Barchart produced by Program 15.7*

Program 15.7 *An applet for drawing bar charts*

```
import java.awt.*;
import java . applet.*;
public class BarChart extends Applet
{
    int n = 0;
    String label [ ] ;
    int value [ ] ;
    public void init ( )
    {
        try
        {
            n = Integer.parseInt (getParameter ("columns") ) ;
            label = new String [n] ;
            value = new int [n] ;

            label [0] = getParameter ("label1") ;
            label [1] = getParameter ("label2") ;
            label [2] = getParameter ("label3") ;
            label [3] = getParameter ("label4") ;

            value [0] = Integer.parseInt (getParameter ("c1") ) ;
            value [1] = Integer.parseInt (getParameter ("c2") ) ;
            value [2] = Integer.parseInt (getParameter ("c3") ) ;
            value [3] = Integer.parseInt (getParameter ("c4") ) ;
        }
        catch (NumberFormatException e) { }
    }
    public void paint (Graphics g)
    {
        for (int i = 0; i < n; i++)
        {
            g.setColor (Color.red) ;
            g.drawstring (label [i], 20,i*50+30) ;
            g.fillRect (50, i*50+10, value [i], 40) ;
        }
    }
}
```

Program 15.8 *HTML file for running the BarChart applet*

```
<HTML>
    <APPLET
    CODE = BarChart.class
    WIDTH = 300
    HEIGHT = 250>

    <PARAM NAME = "columns" VALUE = "4">

    <PARAM NAME = "c1" VALUE = "110">
    <PARAM NAME = "c2" VALUE = "150">
    <PARAM NAME = "c3" VALUE = "100">
    <PARAM NAME = "c4" VALUE = "170">
    <PARAM NAME = "label1" VALUE = "91">
    <PARAM NAME = "label2" VALUE = "92">
    <PARAM NAME = "label3" VALUE = "93">
    <PARAM NAME = "label4" VALUE = "94">

    </APPLET>
</HTML>
```

15.10 INTRODUCTION TO AWT PACKAGE

The Abstract Window Toolkit (AWT) package in Java enables the programmers to create GUI-based applications. It contains a number of classes that help to implement common Windows-based tasks, such as manipulating windows, adding scroll bars, buttons, list items, text boxes, etc. All the classes are contained in the **java.awt** package. These classes are hierarchically arranged inside the awt package in such a manner that each successive level in the hierarchy adds certain attributes to the GUI application.

AWT provides support for both standard and applet windows. Figure 15.13 shows how their corresponding classes are hierarchically arranged in the awt package.

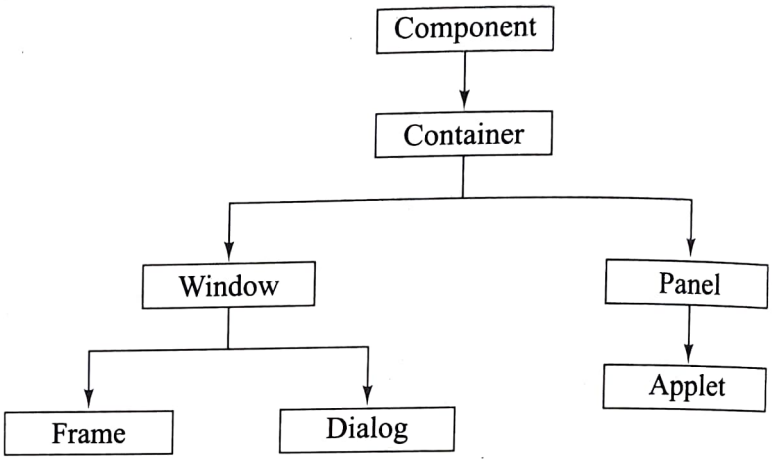


Fig. 15.13 *AWTHierarchy*

Component

Component class is the super class to all the other classes from which various GUI elements are realized. It is primarily responsible for effecting the display of a graphic object on the screen. It also handles the various keyboard and mouse events of the GUI application.

Container

As the name suggests, the **Container** object contains the other awt components. It manages the layout and placement of the various awt components within the container. A container object can contain other containers objects as well; thus allowing nesting of containers.

Window

The **Window** object realizes a top-level window but without any border or menu bar. It just specifies the layout of the window. A typical window that you would want to create in your application is not normally derived from the **Window** class but from its subclass, i.e., **Frame**.

Panel

The super class of applet, **Panel** represents a window space on which the application's output is displayed. It is just like a normal window having no border, title bar, menu bar, etc. A panel can contain within itself other panels as well.

Frame

The **Frame** object realizes a top-level window complete with border and menu bar. It supports common window-related events such as close, open, activate, deactivate, etc. Almost all the programs that we created while discussing applets and graphics programming used one or more classes of the awt package.

15.11 INTRODUCTION TO SWINGS

Similar to AWT, **swing** is also a GUI toolkit that facilitates the creation of highly interactive GUI applications. However, swing is more flexible and robust when it comes to implementing graphical

components. One of the main differences between swing and **awt** is that swing will always generate similar type of output irrespective of the underlying platform. AWT on the other hand, is more dependent on the underlying operating system for generating the graphic components; thus the output may vary from one platform to another.

Swings can be regarded as more graphically-rich than AWT not only because they provide some entirely new graphical components (like tabbed window and tree structure) but also due to the fact that they have enhanced some of the conventional AWT components (like buttons with both image and text).

Some of the key swing classes are:

- **JApplet**: An extension of Applet class, it is the swing's version of applet.
- **JFrame**: An extension of **java.awt.Frame** class, it is the swing's version of frame.
- **JButton**: Helps to realize a push button in swing.
- **JTabbedPane**: Helps to realize a tabbed pane in swing.
- **Jtree**: Helps to realize a hierarchical (tree) structure in swing.
- **JComboBox**: Helps to realize a combo box in swing.

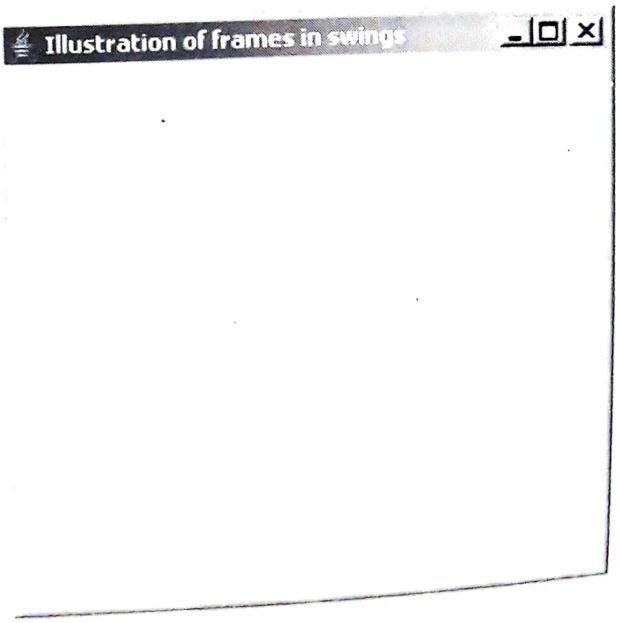
Program 15.9 *A simple program to create a frame object in swing*

```
import javax.swing.*;
public class eg_swing extends JFrame
{
  public static void main(String args[])
{    new eg_swing();  }

  public eg_swing() {
    this.setTitle("Illustration of frames in swings");
    this.setSize(300, 300);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    this.setVisible(true);
  }
}
```

Output of Program 15.9:



15.12 SUMMARY

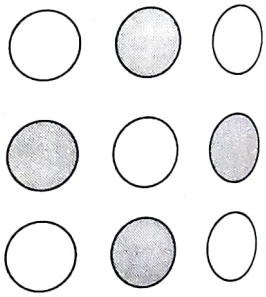
Java's **Graphics** class supports many methods that enable us to draw many types of shapes, including lines, rectangles, ovals, and arcs. We can use these methods to enhance the appearance of outputs of applets, to draw frames around objects, and to put together simple illustrations. We have also seen how the graphics capability of Java can be used to draw line graphs and bar charts.

KEY TERMS

Canvas, Coordinate system, Line graph, Bar graph, Polygon, Arc.

REVIEW QUESTIONS

15.1 How is Java's coordinate system organized?
15.2 Describe the arguments used in the method **drawRoundRect( )**.
15.3 Explain the purpose of each argument used in the method **drawArc( )**.
15.4 Describe the three ways of drawing polygons.
15.5 Write applets to draw the following shapes:
(a) Cone
(b) Cylinder
(c) Cube
(d) Square inside a circle
(e) Circle inside a square



15.6 Write an applet to display the following figure:
15.7 Suggest possible improvements in the human face drawn by Program 15.3.
15.8 Modify the Program 15.8 such that the display for each year would include the year, bar graph and the value represented by the bar as shown below

Year 91 [ ] 110

DEBUGGING EXERCISES

15.1 Find errors in the following code which draws an arc.

```
import java.awt.*;
import java.applet.*;
public class arc extends Applet
{
     public void paint(Graphics g)
     {
          g.drawArc(60,125,80,40,180,180);
          g.fillArc(60,125,80,40,180,180);
          g.setFont("Arial");
          g.drawString("Arc Example",50,50);
     }
}
```

The following code draws a line. Will the code work? If not, why?

```
import java.awt.*;
import java.applet.*;
class line extends Applet
{
public void paint(Graphics g)
     {
          g.drawLine(10,10,100,100,45);
     }
}
```

5.3 The program given below should create a square using drawLine() method. Does the code show the desired output?

```
import java.awt.*;
import java.applet.*;
public class lineSquare extends Applet
{
    public void paint(Graphics g)
    {
         g.drawLine(10,10,10,200);
         g.drawLine(10,10,200,10);
         g.drawLine(10,200,200,200);
         g.drawLine(200,10,200,200);
```

15.4 An oval, line and rectangle are being created using this code. Correct the code.

```
import java.awt.*;
import java.applet.*;
class roundrect extends Applet
{
    public void paint(Graphics g)
    {
         g.drawOval(20,20,120,120) ;
         g.drawLine(1O,1O,1OO,1OO);
         g.drawRect(10,10,100,100,10,50);
    }
}
```

15.5 The following code is designed to draw a rectangle with pink color. What is wrong with the code?

```
import java.awt.*;
import java.applet.*;
public class colorrect extends Applet
{
     public void paint(Graphics g)
     {
          Color c=g.getColor( ) ;
          g.setColor(pink);
          g.drawOval(20,20,100,100) ;
          g.setColor(c) ;
          g.fillRect(120,120,50,50);
     }
}
```