A

PROJECT REPORT

ON

“SCIENTIFIC CALCULATER”

Table of contents:-

1. Front Page

2. Certification

3. Declaration

4. Certificate of Originality

5. Acknowledgement

6. Table of contents

7. Team Information

8. Introduction

9. Objectives

10. Tools/Environment used

11. Analysis document

12. Design Document

13. Project Code

14. Testing

15. Input/Output Screen

16. Implementation of security for the s/w develop

17. Limitation

18. Future Application of Project

19.Bibliography

**INTRODUCTION**

This project is implement in JAVA. This project covers all the arithematic calculations

very effectively.This project is very effective,interactive and can be used easily by any user.

Using this project we can calculate any type of arithematic operation like addition of any

given numbers,subtractions of any given numbers,inversen angle,factorial and log of any

given number etc.

This project is developed using java applet and awt controls in java which make the

project more user friendly and interactive to use.Also due to use of these tools the

presentation of the project becomes attective and it also make it different from other

such application developed previously.

Firstly it is neccessory that we know about the basic of java language because this project is

implemented in java(object oriented language).

**JAVA Language Basics**

## ***Object-Oriented Programming Concepts***

If you've never used an object-oriented language before, you need to understand the underlying concepts before you begin writing code. You need to understand what an object is, what a class is, how objects and classes are related, and how objects communicate by using messages. The first few sections of this chapter describe the concepts behind object-oriented programming. The last section shows how these concepts translate into code.

### *What Is an Object?*

An object is a software bundle of related variables and methods. Software objects are often used to model real-world objects you find in everyday life.

### *What Is a Message?*

Software objects interact and communicate with each other using messages.

### *What Is a Class?*

A class is a blueprint or prototype that defines the variables and the methods common to all objects of a certain kind.

### *What Is Inheritance?*

A class inherits state and behavior from its superclass. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs.

### *What Is an Interface?*

An interface is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface.

## ***Language Basics***

### *Variables*

You use variables in your program to hold data. This section discusses data types, how to initialize variables, and how to refer to variables within blocks of code.

### *Operators*

This section details how you perform various operations, such as arithmetic and assignment operations.

### *Expressions, Statements, and Blocks*

This section discusses how to combine operators and variables into sequences known as expressions. Expressions are the building blocks of your code. You will also learn how to construct statements and statement blocks.

### *Control Flow Statements*

Programs use control flow statements to conditionally execute statements, to loop over statements, or to jump to another area in the program. This section shows you how to control your program's flow with such statements as if-else and while.

## ***Characters and Strings***

Use a Character object to contain a single character value, use a String object to contain a sequence of characters that won't change, and use a StringBuilder or StringBuffer object to construct or to modify a sequence of characters dynamically.

## ***Numbers***

You use an instance of one of the Number classes—Byte, Double, Float, Integer, Long, and Short—to contain a number of primitive type. You can also use BigInteger and BigDecimal for arbitrary-precision numbers.

The Number classes include class methods and constants, which are useful in a variety of ways. The MIN\_VALUE and MAX\_VALUE constants contain the smallest and largest values that can be contained by an object of that type. The byteValue, shortValue, and similar methods convert one numeric type to another. The valueOf method converts a string to a number, and the toString method converts a number to a string.

To format a number to display to an end user, you use the NumberFormat class in the java.text package. When using NumberFormat, you can get a default format for decimal numbers, percentages, or currency. Or, you can design a custom format using patterns. If you are using JDK 5.0 or later, the printf facility provides yet another means for formatting your output.

The Math class contains a variety of class methods for performing mathematical functions. This class includes the trigonometric functions, such as computing sine, cosine, and so on. Math also includes functions for logarithm calculations, as well as basic arithmetic functions, such as rounding. Finally, Math contains a method, random, for generating random numbers.

## ***Arrays***

An array is a structure that holds multiple values of the same type. The length of an array is established when the array is created (at runtime). After creation, an array is a fixed-length structure.

An array element is one of the values within an array and is accessed by its position within the array.

If you want to store data of different types in a single structure, or if you need a structure whose size can change dynamically, use a Collection implementation, such as ArrayList, instead of an array.

An array is a fixed-length data structure that can contain multiple objects of the same type. An array can contain any type of object, including arrays. To declare an array, you use the type of object that the array can contain and brackets.

The length of the array must be specified when it is created. You can use the new operator to create an array, or you can use an array initializer. Once created, the size of the array cannot change. To get the length of the array, you use the length attribute.

An element within an array can be accessed by its index. Indices begin at 0 and end at the length of the array minus 1.

To copy an array, use the arraycopy method in the System class.

## **Classes and Inheritance**

## ***Creating Classes***

A class definition has two parts: a class declaration and a class body. The class body contains member variables, methods, and constructors for the class. A class uses member variables to contain state and uses methods to implement behavior. You control access to member variables and methods in the same way: by using an access specifier, such as private or public, in the member's declaration. You specify a class member variable or a class method by using the static keyword in the member's declaration. A member that is not declared as static is implicitly an instance member. Class variables are shared by all instance of a class and can be accessed through the class name. Instances of a class get their own copy of each instance variable, which must be accessed through an instance reference.

## Managing Inheritance

## **Except for the Object class, a class has exactly one direct superclass. A class inherits member variables and methods from all its superclasses, whether direct or indirect. A subclass can override methods that it inherits, or it can hide variables or methods that it inherits.**

The Object class is the top of the class hierarchy. All classes are descendants from this class and inherit methods from it. Useful methods inherited from Object include toString, equals, clone, getClass, wait, notify, and notifyAll.

You can prevent a class from being subclassed by using the final keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.

An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods — methods that are declared but not implemented. Subclasses provide the implementations for abstract methods.

## Nested Classes

## **You use nested classes to reflect and to enforce the relationship between two classes. You should define a class within another class when the nested class makes sense only in the context of its enclosing class or when it relies on the enclosing class for its function. For example, a text cursor might make sense only in the context of a text component.**

As a member of its enclosing class, a nested class has a special privilege: It has unlimited access to its enclosing class's members, even if they are declared private. However, this special privilege isn't really special at all. It is fully consistent with the meaning of private and the other access specifiers. The access specifiers restrict access to members for classes *outside* the enclosing class. The nested class is *inside* its enclosing class so that it has access to its enclosing class's members.

Like other members, a nested class can be declared static (or not). A static nested class is called just that: a *static nested class*. A nonstatic nested class is called an *inner class*.

## ***Interfaces and Packages***

## Creating and Using Interfaces

## **An interface defines a protocol of communication between two objects. An interface definition is comprised of a declaration and a body. The interface body contains declarations, but no implementations, for a set of methods. An interface might also contain constant definitions. A class that implements an interface must implement all the methods declared in the interface. An interface name can be used anywhere a type can be used.**

## ***Creating and Using Packages***

To create a package, you put a type (class, interface, enum or annotation) in it. To put a class or an interface into a package, you put a package statement as the first statement in the source file for the class or the interface. The path name of the source and class file of a class or an interface mirrors the name of the package.

To use a class or an interface that's in a different package, you have three choices: You can use the fully qualified name of the class or the interface, you can import the class or the interface, or you can import the entire package of which the class or the interface is a member.

You might have to set your class path so that the compiler and the interpreter can find the source and class files for your classes and interfaces.

OBJECTIVE

The objective of the project is to design and develop a calculater which is used for calculating

all the arithematic operation.

The aim of the project to develop an environment by which the user can easily calculate all

the arithematic operation effectively.

The aim of the project is to present all these functions into attractive and user-friendly manner.

### 

### TOOLS/ENVIRONMENT USED.

The program doesn’t need any special hardware or software to run it can run on minimum hardware requirements. In software part also it only needs an operating system with Java runtime system & java compiler on it.

**1. Hardware requirements**

* + Intel Pentium II Processor 233 Mhz. with min 64 MB RAM

**2. Software Requirements**

* Windows 98 onwards.
* java editor/Compiler&java runtime system.

# ANALYSIS DOCUMENT

# FEASIBILITY STUDY

***Identification of Need:-***

This module can be used by the user for calculating all the arithematic calculation to maintain their work based on these mathematical calculation.Such users should work hours in many places we find the things are still maintained through this project. And so it was decided to go ahead with this project.

Processing systems of today include some basic processes like Allow user to calculate all the arithematic calculation currectly for all their records.The above said module all these functionalities and would help the user and also the administrators of the system a lot in their day-to-day processing.

**prelimary Investigation**

Preliminary investigation yielded the following information.

* There are two of the system.
* Administrators and user who work on computer.
* There will be choices shown to the user.
* Based upon their choice the program would be processed.

**Feasibility Study**

Feasibility analysis is a measure of how beneficial or practical the development of software system will be to an organization.

The feasibility analysis takes an insight into the Problem Analysis and the requirement Specification are based on this study**.**

**Problem Definition**

The main problem is to develop a secure, efficient and reliable calculater that should not be easy to distroy and that allows to perform all scientific calculations . The calculater must be strong enough for calculating the scientific and arithmetic calculations.

All the solutions are studied keeping in mind the following types of feasibilities.

**Technical Feasibility**

To study about the technical feasibility of this project we would first take quick look into how it has been developed and what languages have been used in it (Discussed in detail later under S/W and H/W requirements topic).

The project has been developed using java programming language that is one of the most well-known and widely known programming languages all over. From technical point of view any programming language would require a compiler and java runtime system.If these tools are available we can compile and execute the java program. We can compile the java program in command prompt by using **javac** keyword .when we use this keyword a bytecode is generated .This bytecode is stored in a class file.After the generation of byte code we can execute this code by using **java** keyword.

Some times we may find things that are compiler dependent and so we have not used any of these components which might create any such problem and thus the project would give same output when run on any of these compilers.

To end discussion about Technical feasibility I would like to say that the project has been prepared keeping in mind the end users and they require very little knowledge of the processing going on at the backend. And all the three compilers discussed above are easily available and easy to use.

**Economical feasibility**

Economical feasibility wise this project would require very less recourses. Since the compilers are easily available and would not cost much it is economically feasible. Moreover use of this project requires minimum knowledge of computers and anyone having it can handle the software easily and so there no requirement of any computer professionals to handle the same. Also to execute this project there no extra software requirement, which makes it more economically feasible.

To prepare the whole module also there were no extra requirements in economical terms. For example the text editors provided with the compilers are enough to edit the code and run it again and again.

**Operational Feasibility**

There are many operations that the module performs before it generates the output. To talk about the operational feasibility, the project is has been designed to take care of all aspects of the Transaction operations. For example, for administrators there are different operations than that of from customers and visitors. Based upon the choice given as input by the user this process would fetch the required data from the database and show it to the user. For example if a user wants to calculate the factorial number of any given number, he would give choice 2 as the input, and the output will shown on the calculater screen.

All the operations are interdependent, as malfunctioning of any one process would lead to chaotic results. So the project has been made keeping this concept in mind and in no case it would lead to such malfunctioning. For example if we want to calculate the addition of any given number a customer will be able to access them and many more such dependent operations have been successfully implemented in this project.

## **SOFTWARE ENGRNEERING PARADIGM**

***What is a Software Engineering Paradigm?***

An approach to software design and programming is called Software Engineering Paradigm.

Different types of Paradigm are as follows:

* **Object Orientation (3 facts)** - A near-synonym for the object-oriented paradigm
* **Object-Oriented paradigm (7 facts)** - An approach to software design and programming in which software is primarily thought of as a collection of classes that each have responsibilities for various operations, and which are instantiated at run time to create objects
* **Procedural paradigm (8 facts)** - An approach to software design and programming in which software is primarily thought of as a hierarchy of procedures - the root of the hierarchy is typically a main procedure, which calls other procedures, etc. (in contrast to the object-oriented paradigm)

**Object-Oriented paradigm**

An approach to [software](http://www.site.uottawa.ca:4321/oose/software.html) [design](http://www.site.uottawa.ca:4321/oose/design.html) and [programming](http://www.site.uottawa.ca:4321/oose/programming.html) in which [software](http://www.site.uottawa.ca:4321/oose/software.html) is primarily thought of as a collection of classes that each have responsibilities [for](http://www.site.uottawa.ca:4321/oose/for.html) various operations, and which are instantiated at run time to create objects

**has feature** [polymorphism](http://www.site.uottawa.ca:4321/oose/polymorphism.html)

**has fundamental units** objects

**helps** to ensure [communicational cohesion](http://www.site.uottawa.ca:4321/oose/communicationalcohesion.html)

**is a kind of** [paradigm](http://www.site.uottawa.ca:4321/oose/paradigm.html)

**organizes** [code](http://www.site.uottawa.ca:4321/oose/code.html) into classes that each contain procedures [for](http://www.site.uottawa.ca:4321/oose/for.html) manipulating instances of that [class](http://www.site.uottawa.ca:4321/oose/class.html) alone

### procedural paradigm

### 

An approach to [software](http://www.site.uottawa.ca:4321/oose/software.html) [design](http://www.site.uottawa.ca:4321/oose/design.html) and [programming](http://www.site.uottawa.ca:4321/oose/programming.html) in which [software](http://www.site.uottawa.ca:4321/oose/software.html) is primarily thought of as a [hierarchy](http://www.site.uottawa.ca:4321/oose/hierarchy.html) of procedures - the root of the [hierarchy](http://www.site.uottawa.ca:4321/oose/hierarchy.html) is typically a main [procedure](http://www.site.uottawa.ca:4321/oose/procedure.html), which calls other procedures, etc. (in contrast to the [object-oriented paradigm](http://www.site.uottawa.ca:4321/oose/object-orientedparadigm.html))

**hides** [many](http://www.site.uottawa.ca:4321/oose/many.html) of the details of computations

**is a kind of** [paradigm](http://www.site.uottawa.ca:4321/oose/paradigm.html)

**organizes** [code](http://www.site.uottawa.ca:4321/oose/code.html) into procedures that each manipulate different types of data

**provides** [procedural abstraction](http://www.site.uottawa.ca:4321/oose/proceduralabstraction.html)

**works badly** if the [program](http://www.site.uottawa.ca:4321/oose/program.html)'s purpose is to perform calculations on complex data

**works well** if the [program](http://www.site.uottawa.ca:4321/oose/program.html)'s purpose is to perform complex calculations with relatively simple data

]n this project I have applied the second Software Engineering paradigm namely

**Procedural Paradigm.**

Since there are in all around 70 procedures that have been implemented in the project each doing a different job and is being called for a particular task. Here the root procedure is main () function, which then calls the process Menu () function that asks for the choice of the user and then calls do Login () function if the choice obtained earlier is 1. Here the whole code has been organized into different procedures and all are interconnected with each other. And our code is expected to do well since we performing relatively complex calculations with simpler data.

**Waterfall model:**

It suggest a sequential approach to software development that begins at the system level and progresses through analysis , design , coding , testing and implementation

TEST&IMPLEMENTATION

CODE

DESIGN

ANALYSIS

**ANALYSIS:**

The requirement gethering process is intensified and focused especially on software. To

understand the nature of the programs to be built , the software analyst must see the

information domain for the software , as well as required function , behavior , performance

and interface .

### DFD at 0 level

Input choice

Output

Or data

**DESIGN:**

Software design is a multistep process that focuses on four distinct attributes programs

> Data structure

>Software architecture

>Interface representation

>procedural details.

The design process transalates requirements into a representation of the software that can be accessed for quality before coding begins

**CODE GENERATION:**

The design must be translated into a machine readable form. The code generation steps

performs this task

**TESTING :**

Once the code has been generated , program testing begins. The testing process focuses on the logicals internals of the software , ensuring that all the statements have been tested and on the logical internals of the software , ensuring that all the statements have been tested.

***SYSTEM DESIGN:-***

Design is the first step in the development phase for and engineered product or system and of moving from the problem domain towards the solution domain. Design is essentially a bridge between the requirement specification and the final solution of satisfying the requirement.

The design process for software system often has two levels. In the first level, focus is on deciding which models are needed for the system, the specification of each model and how the models should be interconnected. This is called the system design or the top – level design. The second level is the internal logic or design of the models or how the specifications of the models can be satisfied is decided upon the design level is often called detailed design or logical design.

### Process Logic

The process logic of the program is very simple. It is a menu-based application and all the functions are listed in the form of menu items on the screen and the user has to only select the relevant option from the menu. The project is developed in a structured manner using properties of OOP in java. The program is divided into modules to make it easy to understand and develop. Each module is developed separately so that it can be reused in other applications in future

For design the scientific calculater we know about the different tool which are used in this

Project.these tool are as follow-

1. Applet
2. Events
3. AWT Controls

**Applet**

**Applet** class, which provides the necessary support for applets. In Chapter 12, you

were introduced to the general form of an applet and the steps necessary to compile

and run one. In this chapter, we will look at applets in detail.

The **Applet** class is contained in the **java.applet** package. **Applet**

contains several methods that give you detailed control over the execution of your

applet. In addition , **java.applet** also defines three interfaces : **AppletContext**,

**AudioClip**, and **AppletStub**.

***Applet Basics***

All applets are subclasses of **Applet**. Thus, all applets must import **java.applet**.

Applets must also import **java.awt**. Recall that AWT stands for the Abstract Window

Toolkit.

Since all applets run in a window, it is necessary to include support for that window.

Applets are not executed by the console-based Java run-time interpreter. Rather,

they are executed by either a Web browser or an applet viewer. The figures shown in

this chapter were created with the standard applet viewer, called **appletviewer**,

provided by the SDK. But you can use any applet viewer or browser you like.

Execution of an applet does not begin at **main( )**. Actually, few applets even have

**main( )** methods. Instead, execution of an applet is started and controlled with an

entirely different mechanism, which will be explained shortly. Output to your applet’s

window is not performed by **System.out.println( )**. Rather, it is handled with various

AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y

location. Input is also handled differently than in an application.

Once an applet has been compiled, it is included in an HTML file using the

APPLET tag. The applet will be executed by a Java-enabled web browser when it

encounters the APPLET tag within the HTML file. To view and test an applet more

conveniently, simply include a comment at the head of your Java source code file

that contains the APPLET tag. This way, your code is documented with the

necessary HTML statements needed by your applet, and you can test the compiled

applet by starting the applet viewer with your Java source code file specified as the

target. Here is an example of such a comment:

/\*

<applet code="MyApplet" width=200 height=60>

</applet>

\*/

This comment contains an APPLET tag that will run an applet called **MyApplet** in a

window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET

command makes testing applets easier, all of the applets shown in this book will

contain the appropriate APPLET tag embedded in a comment.

***The Applet Class***

The **Applet** class defines the methods shown in Table . **Applet** provides all

necessary support for applet execution, such as starting and stopping. It also

provides methods that load and display images, and methods that load and play

audio clips.

**Applet** extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which

extends **Component**. These classes provide support for Java’s window-based,

graphical interface. Thus, **Applet** provides all of the necessary support for window-

based activities. (The AWT is described in detail in following chapters.)

THE JAVA LIBRARY

Method Description

void destroy( ) Called by the browser just before

an applet is terminated. Your

applet will override this method if it

needs to perform any cleanup prior to

its destruction.

void init( ) Called when an applet begins execution.

void start( ) Called by the browser when an applet

should start (or resume) execution. It is

automatically called after **init( )** when

an applet first begins.

void stop( ) Called by the browser to suspend

execution of the applet. Once stopped,

an applet is restarted when the browser

***Applet Initialization and Termination***

It is important to understand the order in which the various methods shown in the

skeleton are called. When an applet begins, the AWT calls the following methods, in

this sequence:

1. **init( )**

2. **start( )**

3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**

2. **destroy( )**

Let’s look more closely at these methods.

***init( )***

The **init( )** method is the first method to be called. This is where you should initialize

variables. This method is called only once during the run time of your applet.

***start( )***

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has

been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )**

is called each time an applet’s HTML document is displayed onscreen. So, if a user

leaves a web page and comes back, the applet resumes execution at **start( )**.

***paint( )***

The **paint( )** method is called each time your applet’s output must be redrawn. This

situation can occur for several reasons. For example, the window in which the applet

is running may be overwritten by another window and then uncovered. Or the applet

window may be minimized and then restored. **paint( )** is also called when the applet

begins execution. Whatever the cause, whenever the applet must redraw its output,

**paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This

parameter will contain the graphics context, which describes the graphics

environment in which the applet is running. This context is used whenever output to

the applet is required.

***stop( )***

The **stop( )** method is called when a web browser leaves the HTML document

Containing the applet—when it goes to another page, for example. When **stop( )** is

called, the applet is probably running. You should use **stop( )** to suspend threads

that don’t need to run when the applet is not visible. You can restart them when

**start( )** is called if the user returns to the page.

***destroy( )***

The **destroy( )** method is called when the environment determines that your applet

needs to be removed completely from memory. At this point, you should free

up any resources the applet may be using. The **stop( )** method is always called

before **destroy( )**.

Simple Applet Display Methods

As we’ve mentioned, applets are displayed in a window and they use the AWT to

perform input and output. Although we will examine the methods, procedures, and

techniques necessary to fully handle the AWT windowed environment in subsequent

chapters, a few are described here, because we will use them to write sample

applets.

to output a string to an applet, use **drawString( )**,

which is a member of the **Graphics** class. Typically, it is called from within either

**update( )** or **paint( )**. It has the following general form:

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

Here, *message* is the string to be output beginning at *x,y.* In a Java window, the

upper-left corner is location 0,0. The **drawString( )** method will not recognize

newline characters. If you want to start a line of text on another line, you must do so

manually, specifying the precise X,Y location where you want the line to begin. (

To set the background color of an applet’s window, use **setBackground( )**. To set

the foreground color (the color in which text is shown, for example), use

**setForeground( )**.

These methods are defined by **Component**, and they have the following general

forms:

void setBackground(Color *newColor*)

void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class **Color** defines the constants

shown here that can be used to specify colors:

Color.black Color.magenta

Color.blue Color.orange

Color.cyan Color.pink

Color.darkGray Color.red

Color.gray Color.white

Color.green Color.yellow

Color.lightGray

For example, this sets the background color to green and the text color to red:

setBackground(Color.green);

setForeground(Color.red);

THE JAVA LIBRARY

A good place to set the foreground and background colors is in the **init( )** method.

Of course, you can change these colors as often as necessary during the execution

of your applet. The default foreground color is black. The default background color is

light gray.

You can obtain the current settings for the background and foreground colors by

calling **getBackground( )** and **getForeground( )**, respectively. They are also

defined by **Component** and are shown here:

Color getBackground( )

Color getForeground( )

***Events***

In the delegation model, an *event* is an object that describes a state change in a

source.

It can be generated as a consequence of a person interacting with the elements in a

graphical user interface. Some of the activities that cause events to be generated are

pressing a button, entering a character via the keyboard, selecting an item in a list,

and clicking the mouse. Many other user operations could also be cited as

examples.

Events may also occur that are not directly caused by interactions with a user

interface.

For example, an event may be generated when a timer expires, a counter exceeds a

value, a software or hardware failure occurs, or an operation is completed. You are

free to define events that are appropriate for your application.

***Event Sources***

A *source* is an object that generates an event. This occurs when the internal state of

that object changes in some way. Sources may generate more than one type of

event.

A source must register listeners in order for the listeners to receive notifications

about a specific type of event. Each type of event has its own registration method.

Here is the general form:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For

example,the method that registers a keyboard event listener is called

**addKeyListener()**

The method that registers a mouse motion listener is called

**addMouseMotionListener( ).**

When an event occurs, all registered listeners are notified and receive a copy of the

Event object. This is known as *multicasting* the event. In all cases, notifications are

sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such

a method is this:

public void add*Type*Listener(*Type*Listener *el*)

throws java.util.TooManyListenersException

THE JAVA LIBRARY

Here, *Type* is the name of the event and *el* is a reference to the event listener. When

such an event occurs, the registered listener is notified. This is known as *unicasting*

the event.

A source must also provide a method that allows a listener to unregister an interest

in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For

example,to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates

events. For example, the **Component** class provides methods to add and remove

keyboard and mouse event listeners.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major

requirements.

First, it must have been registered with one or more sources to receive notifications

about specific types of events. Second, it must implement methods to receive and

process these notifications.

The methods that receive and process events are defined in a set of interfaces found

in **java.awt.event**. For example, the **MouseMotionListener** interface defines two

methods to receive notifications when the mouse is dragged or moved. Any object

may receive and process one or both of these events if it provides an implementation

of this interface.

***Event Classes***

The classes that represent events are at the core of Java’s event handling

mechanism.

Thus, we begin our study of event handling with a tour of the event classes. As you

will see, they provide a consistent, easy-to-use means of encapsulating events.

At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**.

It is the superclass for all events. Its one constructor is shown here:

EventObject(Object *src*)

Here, *src* is the object that generates this event.

**EventObject** contains two methods: **getSource( )** and **toString( )**. The **getSource( )**

method returns the source of the event. Its general form is shown here:

Object getSource( )

As expected, **toString( )** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of

**EventObject**. It is the superclass (either directly or indirectly) of all AWT-based

events used by the delegation event model. Its **getID( )** method can be used to

determine the type of the event. The signature of this method is shown here:

int getID( )

Additional details about **AWTEvent** are provided at the end of Chapter 22. At this

point, it is important to know only that all of the other classes discussed in this

section are subclasses of **AWTEvent**.

To summarize:

■ **EventObject** is a superclass of all events.

■ **AWTEvent** is a superclass of all AWT events that are handled by the delegation

event model.

The package **java.awt.event** defines several types of events that are generated by

various user interface elements. Table enumerates the most important of these

event classes and provides a brief description of when they are generated. The most

commonly used constructors and methods in each class are described in the

following sections.

THE JAVA LIBRARY

***Event Class Description***

ActionEvent Generated when a button is pressed, a list item is

double-clicked, or a menu item is selected.

AdjustmentEvent Generated when a scroll bar is manipulated.

ComponentEvent Generated when a component is hidden, moved, resized,

or becomes visible.

ContainerEvent Generated when a component is added to or removed

from a container.

FocusEvent Generated when a component gains or loses

keyboard focus.

InputEvent Abstract super class for all component input event classes.

ItemEvent Generated when a check box or list item is clicked; also

occurs when a choice selection is made or a checkable

menu item is selected or deselected.

***The KeyEvent Class***

A **KeyEvent** is generated when keyboard input occurs. There are three types

of key events, which are identified by these integer constants: **KEY\_PRESSED**,

**KEY\_RELEASED**, and **KEY\_TYPED**. The first two events are generated when any

key is pressed or released. The last event occurs only when a character is

generated.

Remember, not all key presses result in characters. For example, pressing the

SHIFT key does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example,

**VK\_0** through **VK\_9** and **VK\_A** through **VK\_Z** define the ASCII equivalents of the

numbers and letters. Here are some others:

**VK\_ENTER VK\_ESCAPE VK\_CANCEL VK\_UP**

**VK\_DOWN VK\_LEFT VK\_RIGHT VK\_PAGE\_DOWN**

**VK\_PAGE\_UP VK\_SHIFT VK\_ALT VK\_CONTROL**

THE JAVA LIBRARY

The **VK** constants specify *virtual key codes* and are independent of any modifiers,

such as control, shift, or alt.

**KeyEvent** is a subclass of **InputEvent**. Here are two of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*)

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the

Event is specified by *type*. The system time at which the key was pressed is passed

in *when*.

The *modifiers* argument indicates which modifiers were pressed when this key

event occurred.

The virtual key code, such as **VK\_UP**, **VK\_A**, and so forth, is passed in *code*. The

character equivalent (if one exists) is passed in *ch*. If no valid character exists, then

*ch* contains **CHAR\_UNDEFINED**. For **KEY\_TYPED** events, *code* will contain

**VK\_UNDEFINED**.

The **KeyEvent** class defines several methods, but the most commonly used ones

are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode(**

**)**,

which returns the key code. Their general forms are shown here:

char getKeyChar( )

int getKeyCode( )

If no valid character is available, then **getKeyChar( )** returns **CHAR\_UNDEFINED**.

When a **KEY\_TYPED** event occurs, **getKeyCode( )** returns **VK\_UNDEFINED**.

The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following

integer constants that can be used to identify them:

MOUSE\_CLICKED The user clicked the mouse.

MOUSE\_DRAGGED The user dragged the mouse.

MOUSE\_ENTERED The mouse entered a component.

MOUSE\_EXITED The mouse exited from a component.

MOUSE\_MOVED The mouse moved.

MOUSE\_PRESSED The mouse was pressed.

MOUSE\_RELEASED The mouse was released.

MOUSE\_WHEEL The mouse wheel was moved (Java 2, v1.4).

**MouseEvent** is a subclass of **InputEvent**. Here is one of its constructors.

MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,

int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here, *src* is a reference to the component that generated the event. The type of the

event is specified by *type*. The system time at which the mouse event occurred is

passed in *when*. The *modifiers* argument indicates which modifiers were pressed

when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*.

The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event

causes a pop-up

menu to appear on this platform. Java 2, version 1.4 adds a second constructor

which also allows the button that caused the event to be specified.

The most commonly used methods in this class are **getX( )** and **getY( )**. These

return the X and Y coordinates of the mouse when the event occurred. Their forms

are shown here:

int getX( )

int getY( )

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the

mouse. It is shown here:

Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and y.

The **translatePoint( )** method changes the location of the event. Its form is shown here:

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.

The **getClickCount( )** method obtains the number of mouse clicks for this event.

Its signature is shown here:

int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear

on this platform. Its form is shown here:

boolean isPopupTrigger( )

Java 2, version 1.4 added the **getButton( )** method, shown here.

int getButton( )

It returns a value that represents the button that caused the event. The return value

will be one of these constants defined by **MouseEvent**.

NOBUTTON BUTTON1 BUTTON2 BUTTON3

The **NOBUTTON** value indicates that no button was pressed or released.

***Sources of Events***

Table lists some of the user interface components that can generate the events

described in the previous section. In addition to these graphical user interface

elements, other components, such as an applet, can generate events. For example,

you receive key and mouse events from an applet. (You may also build your own

components that generate events.) In this chapter we will be handling only mouse

and keyboard events, but the following two chapters will be handling events from the

sources shown in Table .

***Event Source Description***

Button Generates action events when the button is pressed.

Checkbox Generates item events when the check box is selected or deselected.

Choice Generates item events when the choice is changed.

List Generates action events when an item is double-clicked; generates

item events when an item is selected or deselected.

Menu Item Generates action events when a menu item is selected; generates item

events when a checkable menu item is selected or deselected.

Scrollbar Generates adjustment events when the scroll bar is manipulated.

Text components Generates text events when the user enters a character.

Window Generates window events when a window is activated, closed,

deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners.

Listeners are created by implementing one or more of the interfaces defined by the

**java.awt.event** package. When an event occurs, the event source invokes the

appropriate method defined by the listener and provides an event object as its

argument. Table lists commonly used listener interfaces and provides a brief

description of the methods that they define.

The following sections examine the specific methods that are contained in each

interface.

THE JAVA LIBRARY

***Interface Description***

ActionListener Defines one method to receive action events.

AdjustmentListener Defines one method to receive adjustment events.

ComponentListener Defines four methods to recognize when a component is

hidden, moved, resized, or shown.

ContainerListener Defines two methods to recognize when a component is

added to or removed from a container.

FocusListener Defines two methods to recognize when a component gains

or loses keyboard focus.

ItemListener Defines one method to recognize when the state of an

item changes.

KeyListener Defines three methods to recognize when a key is

released, or typed.

MouseMotionListener Defines two methods to recognize when the mouse is

dragged or moved.

MouseWheelListener Defines one method to recognize when the mouse wheel is

moved. (Added by Java 2, version 1.4)

TextListener Defines one method to recognize when a text value changes.

WindowFocusListener Defines two methods to recognize when a window gains or

loses input focus. (Added by Java 2, version 1.4)

WindowListener Defines seven methods to recognize when a window is

activated, closed, deactivated, deiconified, iconified, opened,

or quit.

***Handling Keyboard Events***

To handle keyboard events, you use the same general architecture as that shown in

the mouse event example in the preceding section. The difference, of course, is that

you will be implementing the **KeyListener** interface.

Before looking at an example, it is useful to review how key events are generated.

When a key is pressed, a **KEY\_PRESSED** event is generated. This results in a call

to the **keyPressed( )** event handler. When the key is released, a **KEY\_RELEASED**

event is generated and the **keyReleased( )** handler is executed. If a character is

generated by the keystroke, then a **KEY\_TYPED** event is sent and the **keyTyped( )**

handler is invoked.

Thus, each time the user presses a key, at least two and often three events are

generated.

If all you care about are actual characters, then you can ignore the information

passed by the key press and release events. However, if your program needs to

handle special keys,such as the arrow or function keys, then it must watch for them

through the **keyPressed( )**handler.

There is one other requirement that your program must meet before it can process

keyboard events: it must request input focus. To do this, call **requestFocus( )**, which

is defined by **Component**. If you don’t, then your program will not receive any

keyboard events.

The following program demonstrates keyboard input. It echoes keystrokes to the

applet window and shows the pressed/released status of each key in the status

window.

// Demonstrate the key event handlers.

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="SimpleKey" width=300 height=100>

</applet>

\*/

public class SimpleKey extends Applet

implements KeyListener {

String msg = "";

int X = 10, Y = 20; // output coordinates

public void init() {

addKeyListener(this);

requestFocus(); // request input focus

}

public void keyPressed(KeyEvent ke) {

showStatus("Key Down");

}

public void keyReleased(KeyEvent ke) {

showStatus("Key Up");

}

public void keyTyped(KeyEvent ke) {

msg += ke.getKeyChar();

repaint();

}

// Display keystrokes.

public void paint(Graphics g) {

g.drawString(msg, X, Y);

}

}

THE JAV

***Introducing the AWT: Working with Windows,Graphics, and Text***

The AWT contains numerous classes and methods that allow you to create

and manage windows.

Although the main purpose of the AWT is to support applet windows, it can

also be used to create stand-alone windows that run in a GUI environment, such as

Windows. Most of the examples are contained in applets, so to run them, you need

to use an applet viewer or a Java-compatible Web browser. A few examples will

demonstrate the creation of stand-alone, windowed programs.

AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java’s largest

packages. Fortunately, because it is logically organized in a top-down, hierarchical

fashion, it is easier to understand and use than you might at first believe. Table

lists some of the many AWT classes.

***Class Description***

AWTEvent Encapsulates AWT events.

AWTEvent Multicaster Dispatches events to multiple listeners.

BorderLayout The border layout manager. Border layouts use five

components: North, South, East, West, and Center.

Button Creates a push button control.

Checkbox Creates a check box control.

CheckboxGroup Creates a group of check box controls.

Event Encapsulates events.

FlowLayout The flow layout manager. Flow layout positions

components left to right, top to bottom.

Frame Creates a standard window that has a title bar, resize

corners, and a menu bar.

Label Creates a label that displays a string.

GridLayout The grid layout manager. Grid layout displays

components in a two-dimensional grid.

Panel The simplest concrete subclass of **Container**.

***Window Fundamentals***

The AWT defines windows according to a class hierarchy that adds functionality and

specificity with each level. The two most common windows are those derived from

**Panel**, which is used by applets, and those derived from **Frame**, which creates a

standard window. Much of the functionality of these windows is derived from their

parent classes. Thus, a description of the class hierarchies relating to these two

classes is fundamental to their understanding. Figure 21-1 shows the class hierarchy

for **Panel** and **Frame**. Let’s look at each of these classes now.

***Component***

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract

class that encapsulates all of the attributes of a visual component. All user interface

elements that are displayed on the screen and that interact with the user are

subclasses of **Component**. It defines over a hundred public methods that are

responsible for managing events, such as mouse and keyboard input, positioning

and sizing the window, and repainting. (You already used many of these methods

when you created applets in Chapters 19 and 20.) A **Component** object is

responsible for remembering the current foreground and background colors and the

currently selected text font.

***Container***

The **Container** class is a subclass of **Component**. It has additional methods that

allow other **Component** objects to be nested within it. Other **Container** objects can

be stored inside of a **Container** (since they are themselves instances of

**Component**). This makes for a multileveled containment system. A container is

responsible for laying out (that is, positioning) any components that it contains. It

does this through the use of various layout managers.

***Panel***

The **Panel** class is a concrete subclass of **Container**. It doesn’t add any new

methods;

it simply implements **Container**. A **Panel** may be thought of as a recursively

nestable,concrete screen component. **Panel** is the superclass for **Applet**. When

screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In

essence, a **Panel** is a window that does not contain a title bar, menu bar, or border.

This is why you don’t see these items when an applet is run inside a browser. When

you run an applet using an applet viewer, the applet viewer provides the title and

border.

Other components can be added to a **Panel** object by its **add( )** method (inherited

from **Container**). Once these components have been added, you can position and

resize them manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods

defined by **Component**.

***Window***

The **Window** class creates a top-level window. A *top-level window* is not contained

within any other object; it sits directly on the desktop. Generally, you won’t create

**Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**,

described next.

***Frame***

**Frame** encapsulates what is commonly thought of as a “window.” It is a subclass of

**Window** and has a title bar, menu bar, borders, and resizing corners. If you create a

**Frame** object from within an applet, it will contain a warning message, such as “Java

Applet Window,” to the user that an applet window has been created. This message

warns users that the window they see was started by an applet and not by software

running on their computer. (An applet that could masquerade as a host-based

application could be used to obtain passwords and other sensitive information

without the user’s knowledge.) When a **Frame** window is created by a program

rather than an applet, a normal window is created.

***Creating a Frame Window in an Applet***

While it is possible to simply create a window by creating an instance of **Frame**, you

will seldom do so, because you will not be able to do much with it. For example, you

will not be able to receive or process events that occur within it or easily output

information to it. Most of the time, you will create a subclass of **Frame**. Doing so lets

you override **Frame**’s methods and event handling.

Creating a new frame window from within an applet is actually quite easy. First,

createa subclass of **Frame**. Next, override any of the standard window methods,

such as **init( )**,**start( )**, **stop( )**, and **paint( )**. Finally, implement the **windowClosing(**

**)** method of the **WindowListener** interface, calling **setVisible(false)** when the

window is closed.

Once you have defined a **Frame** subclass, you can create an object of that class.

This causes a frame window to come into existence, but it will not be initially visible.

You make it visible by calling **setVisible( )**. When created, the window is given a

default height and width. You can set the size of the window explicitly by calling the

**setSize( )** method.

The following applet creates a subclass of **Frame** called **SampleFrame**. A window

of this subclass is instantiated within the **init( )** method of **AppletFrame**. Notice that

**SampleFrame** calls **Frame**’s constructor. This causes a standard frame window to

becreated with the title passed in **title**. This example overrides the applet window’s

**start( )** and **stop( )** methods so that they show and hide the child window, respectively.

This causes the window to be removed automatically when you terminate the applet,

when you close the window, or, if using a browser, when you move to another page.

It also causes the child window to be shown when the browser returns to the applet.

// Create a child frame window from within an applet.

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="AppletFrame" width=300 height=50>

</applet>

\*/

// Create a subclass of Frame.

class SampleFrame extends Frame {

SampleFrame(String title) {

super(title);

// create an object to handle window events

MyWindowAdapter adapter = new MyWindowAdapter(this);

// register it to receive those events

addWindowListener(adapter);

}

public void paint(Graphics g) {

g.drawString("This is in frame window", 10, 40);

}

}

class MyWindowAdapter extends WindowAdapter {

SampleFrame sampleFrame;

public MyWindowAdapter(SampleFrame sampleFrame) {

this.sampleFrame = sampleFrame;

}

public void windowClosing(WindowEvent we) {

sampleFrame.setVisible(false);

}

}

// Create frame window.

public class AppletFrame extends Applet {

Frame f;

public void init() {

f = new SampleFrame("A Frame Window");

f.setSize(250, 250);

f.setVisible(true);

}

public void start() {

f.setVisible(true);

}

public void stop() {

f.setVisible(false);

}

public void paint(Graphics g) {

g.drawString("This is in applet window", 10, 20);

}

}

***Handling Events in a Frame Window***

Since **Frame** is a subclass of **Component**, it inherits all the capabilities defined by

**Component**. This means that you can use and manage a frame window that you

Create just like you manage your applet’s main window. For example, you can

override **paint( )** to display output, call **repaint( )** when you need to restore the

window, and override all event handlers. Whenever an event occurs in a window, the

event handlers defined by that window will be called. Each window handles its own

events. For example, the following program creates a window that responds to

mouse events.

***Working with Graphics***

The AWT supports a rich assortment of graphics methods. All graphics are drawn

relative to a window. This can be the main window of an applet, a child window of

an applet, or a stand-alone application window. The origin of each window is at the

top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window

takes place through a graphics context. A *graphics context* is encapsulated by the

**Graphics** class and is obtained in two ways:

■ It is passed to an applet when one of its various methods, such as **paint( )** or

**update( )**, is called.

■ It is returned by the **getGraphics( )** method of **Component**.

The **Graphics** class defines a number of drawing functions. Each shape can be

drawn edge-only or filled. Objects are drawn and filled in the currently selected

graphics color, which is black by default. When a graphics object is drawn that

exceeds the dimensions of the window, output is automatically clipped.

***Working with Fonts***

The AWT supports multiple type fonts. Fonts have emerged from the domain of

traditional typesetting to become an important part of computer-generated

documents and displays. The AWT provides flexibility by abstracting font-

manipulation operations and allowing for dynamic selection of fonts.

Beginning with Java 2, fonts have a family name, a logical font name, and a face

name. The *family name* is the general name of the font, such as Courier. The *logical*

*name* specifies a category of font, such as Monospaced. The *face name* specifies a

specific font,such as Courier Italic.

Fonts are encapsulated by the **Font** class. Several of the methods defined by **Font**

are listed in Table .

The **Font** class defines these variables:

Variable Meaning

String name Name of the font

float pointSize Size of the font in points

int size Size of the font in points

int style Font style

Method Description

static Font decode(String *str*) Returns a font given its name.

boolean equals(Object *FontObj*) Returns **true** if the invoing object contains

the same font as that specified by *FontObj.*

Otherwise, it returns **false**.

String getFamily( ) Returns the name of the font family to

which the invoking font belongs.

static Font getFont(String *property*) Returns the font associated with the system

property specified by *property.* **null** is

returned if *property* does not exist.

String getFontName() Returns the face name of the invoking font.

(Added by Java 2)

String getName( ) Returns the logical name of the

invoking font.

int getSize( ) Returns the size, in points, of the

invoking font.

int getStyle( ) Returns the style values of the invoking font.

int hashCode( ) Returns the hash code associated with the

invoking object.

boolean isBold( ) Returns **true** if the font includes the **BOLD**

style value. Otherwise, **false** is returned.

boolean isItalic( ) Returns **true** if the font includes the **ITALIC**

style value. Otherwise, **false** is returned.

boolean isPlain( ) Returns **true** if the font includes the **PLAIN**

style value. Otherwise, **false** is returned.

String toString( ) Returns the string equivalent of the

invoking font.

***Creating and Selecting a Font***

To select a new font, you must first construct a **Font** object that describes that font.

One **Font** constructor has this general form:

Font(String *fontName*, int *fontStyle*, int *pointSize*)

Here, *fontName* specifies the name of the desired font. The name can be specified

using either the logical or face name. All Java environments will support the following

fonts: Dialog, DialogInput, Sans Serif, Serif, Monospaced, and Symbol. Dialog is the

font used by your system’s dialog boxes. Dialog is also the default if you don’t

explicitly set a font. You can also use any other fonts supported by your particular

environment, but be careful—these other fonts may not be universally available.

The style of the font is specified by *fontStyle.* It may consist of one or more of these

three constants: **Font.PLAIN**, **Font.BOLD**, and **Font.ITALIC**. To combine styles, OR

them together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.

The size, in points, of the font is specified by *pointSize.*

To use a font that you have created, you must select it using **setFont( )**, which is

defined by **Component**. It has this general form:

void setFont(Font *fontObj*)

Here, *fontObj* is the object that contains the desired font.

The following program outputs a sample of each standard font. Each time you click

the mouse within its window, a new font is selected and its name is displayed.

// Show fonts.

import java.applet.\*;

import java.awt.\*;

import java.awt.event.\*;

/\*

<applet code="SampleFonts" width=200 height=100>

</applet>

\*/

public class SampleFonts extends Applet {

int next = 0;

Font f;

String msg;

public void init() {

f = new Font("Dialog", Font.PLAIN, 12);

msg = "Dialog";

setFont(f);

addMouseListener(new MyMouseAdapter(this));

}

public void paint(Graphics g) {

g.drawString(msg, 4, 20);

}

}

class MyMouseAdapter extends MouseAdapter {

SampleFonts sampleFonts;

public MyMouseAdapter(SampleFonts sampleFonts) {

this.sampleFonts = sampleFonts;

}

public void mousePressed(MouseEvent me) {

// Switch fonts with each mouse click.

sampleFonts.next++;

switch(sampleFonts.next) {

case 0:

sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);

sampleFonts.msg = "Dialog";

break;

case 1:

sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);

sampleFonts.msg = "DialogInput";

break;

case 2:

sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);

sampleFonts.msg = "SansSerif";

break;

case 3:

sampleFonts.f = new Font("Serif", Font.PLAIN, 12);

sampleFonts.msg = "Serif";

break;

case 4:

sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);

sampleFonts.msg = "Monospaced";

break;

}

if(sampleFonts.next == 4) sampleFonts.next = -1;

sampleFonts.setFont(sampleFonts.f);

sampleFonts.repaint();

}

}

**AWT Controls:-**

It examines the standard controls and layout managers defined by Java. It also

discusses menus and the menu bar.

*Controls* are components that allow a user to interact with your application in

various ways—for example, a commonly used control is the push button. A *layout*

*manager* automatically positions components within a container. Thus, the

appearance of a window is determined by a combination of the controls that it

contains and thelayout manager used to position them.

In addition to the controls, a frame window can also include a standard-style *menu*

*bar.*

Each entry in a menu bar activates a drop-down menu of options from which the

user Can choose. A menu bar is always positioned at the top of a window. Although

different in appearance, menu bars are handled in much the same way as are the

other controls.

***Control Fundamentals***

The AWT supports the following types of controls:

■ Labels

■ Push buttons

■ Check boxes

■ Choice lists

■ Lists

■ Scroll bars

■ Text editing

These controls are subclasses of **Component**.

***Adding and Removing Controls***

To include a control in a window, you must add it to the window. To do this, you must

first create an instance of the desired control and then add it to a window by calling **add( )**,

which is defined by **Container**. The **add( )** method has several forms. The following

form is the one that is used for the first part of this chapter:

Component add(Component *compObj*)

Here, *compObj* is an instance of the control that you want to add. A reference to

*compObj* is returned. Once a control has been added, it will automatically be visible

whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is

no longer needed. To do this, call **remove( )**. This method is also defined by

**Container**.

It has this general form:

void remove(Component *obj*)

Here, *obj* is a reference to the control you want to remove. You can remove all

controls by calling **removeAll( )**.

***Labels***

The easiest control to use is a label. A *label* is an object of type **Label**, and it

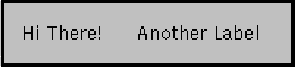
contains a string, which it displays. Labels are passive controls that do not support

any interaction with the user. **Label** defines the following constructors:

Label( )

Label(String *str*)

Label(String *str*, int *how*)



Label

The first version creates a blank label. The second version creates a label that

contains the string specified by *str.* This string is left-justified. The third version

creates a label that contains the string specified by *str* using the alignment specified

by *how.* The value of *how* must be one of these three constants: **Label.LEFT**,

**Label.RIGHT**, or **Label.CENTER**.

You can set or change the text in a label by using the **setText( )** method. You can

obtain the current label by calling **getText( )**. These methods are shown here:

void setText(String *str*)

String getText( )

For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.

You can set the alignment of the string within the label by calling **setAlignment( )**.

To obtain the current alignment, call **getAlignment( )**. The methods are as follows:

void setAlignment(int *how*)

int getAlignment( )

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet:

// Demonstrate Labels

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="LabelDemo" width=300 height=200>

</applet>

\*/

public class LabelDemo extends Applet {

public void init() {

Label one = new Label("One");

Label two = new Label("Two");

Label three = new Label("Three");

// add labels to applet window

add(one);

add(two);

add(three);

}

}

Following is the window created by the **LabelDemo** applet. Notice that the labels are

organized in the window by the default layout manager. Later, you will see how to

control more precisely the placement of the labels.

***Using Buttons***

The most widely used control is the push button. A *push button* is a component that

contains a label and that generates an event when it is pressed. Push buttons are

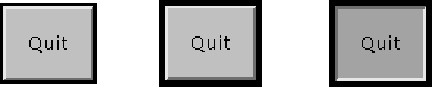
objects of type **Button**. **Button** defines these two constructors:

Button( )

Button(String *str*)

The first version creates an empty button. The second creates a button that contains *str*

as a label.



**Button**

After a button has been created, you can set its label by calling **setLabel( )**. You can

retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String *str*)

String getLabel( )

Here, *str* becomes the new label for the button.

***Handling Buttons***

Each time a button is pressed, an action event is generated. This is sent to any

listeners that previously registered an interest in receiving action event notifications

from that component. Each listener implements the **ActionListener** interface. That

interface defines the **actionPerformed( )** method, which is called when an event

occurs. An **ActionEvent** object is supplied as the argument to this method. It

contains both a reference to the button that generated the event and a reference to

the string that is the label of the button. Usually,either value may be used to identify

the button, as you will see.

Here is an example that creates three buttons labeled “Yes,” “No,” and “Undecided.”

Each time one is pressed, a message is displayed that reports which button has

been pressed. In this version, the label of the button is used to determine which

button has been pressed. The label is obtained by calling the **getActionCommand(**

**)** method on the **ActionEvent** object passed to **actionPerformed( )**.

// Demonstrate Buttons

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="ButtonDemo" width=250 height=150>

</applet>

\*/

public class ButtonDemo extends Applet implements ActionListener

{

String msg = "";

Button yes, no, maybe;

public void init() {

yes = new Button("Yes");

no = new Button("No");

maybe = new Button("Undecided");

add(yes);

add(no);

add(maybe);

yes.addActionListener(this);

no.addActionListener(this);

maybe.addActionListener(this);

}

public void actionPerformed(ActionEvent ae) {

String str = ae.getActionCommand();

if(str.equals("Yes")) {

msg = "You pressed Yes.";

}

else if(str.equals("No")) {

msg = "You pressed No.";

}

else {

msg = "You pressed Undecided.";

}

repaint();

}

public void paint(Graphics g) {

g.drawString(msg, 6, 100);

}

}

***Applying Check Boxes***

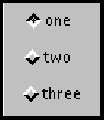
A *check box* is a control that is used to turn an option on or off. It consists of a small

box that can either contain a check mark or not. There is a label associated with

each check box that describes what option the box represents. You change the state

of a check box by clicking on it. Check boxes can be used individually or as part of a

group. Check boxes are objects of the **Checkbox** class.



**Check Box**

**Checkbox** supports these constructors:

Checkbox( )

Checkbox(String *str*)

Checkbox(String *str*, boolean *on*)

Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*)

Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*)

The first form creates a check box whose label is initially blank. The state of the

check box is unchecked. The second form creates a check box whose label is

specified by *str.*

The state of the check box is unchecked. The third form allows you to set the initial

state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is

cleared. The fourth and fifth forms create a check box whose label is specified by *str*

and whose group is specified by *cbGroup.* If this check box is not part of a group,

then *cbGroup* must be **null**. (Check box groups are described in the next section.)

The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**. To set its state, call

**setState( )**. You can obtain the current label associated with a check box by calling

**getLabel( )**. To set the label, call **setLabel( )**. These methods are as follows:

boolean getState( )

void setState(boolean *on*)

String getLabel( )

void setLabel(String *str*)

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string

passed in *str* becomes the new label associated with the invoking check box.

***Handling Check Boxes***

Each time a check box is selected or deselected, an item event is generated. This is

sent to any listeners that previously registered an interest in receiving item event

notifications from that component. Each listener implements the **ItemListener**

interface.

That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is

supplied as the argument to this method. It contains information about the event (for

example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first box is

checked. The status of each check box is displayed. Each time you change the state

of a check box, the status display is updated.

// Demonstrate check boxes.

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="CheckboxDemo" width=250 height=200>

</applet>

\*/

public class CheckboxDemo extends Applet implements ItemListener

{

String msg = "";

Checkbox Win98, winNT, solaris, mac;

public void init() {

Win98 = new Checkbox("Windows 98/XP", null, true);

winNT = new Checkbox("Windows NT/2000");

solaris = new Checkbox("Solaris");

mac = new Checkbox("MacOS");

add(Win98);

add(winNT);

add(solaris);

add(mac);

Win98.addItemListener(this);

winNT.addItemListener(this);

solaris.addItemListener(this);

mac.addItemListener(this);

}

public void itemStateChanged(ItemEvent ie) {

repaint();

}

// Display current state of the check boxes.

public void paint(Graphics g) {

msg = "Current state: ";

g.drawString(msg, 6, 80);

msg = " Windows 98/XP: " + Win98.getState();

g.drawString(msg, 6, 100);

msg = " Windows NT/2000: " + winNT.getState();

g.drawString(msg, 6, 120);

msg = " Solaris: " + solaris.getState();

g.drawString(msg, 6, 140);

msg = " MacOS: " + mac.getState();

g.drawString(msg, 6, 160);

}

}

***Understanding Layout Managers***

All of the components that we have shown so far have been positioned by the default

layout manager. As we mentioned at the beginning of this chapter, a layout manager

automatically arranges your controls within a window by using some type of

algorithm.

If you have programmed for other GUI environments, such as Windows, then you

are accustomed to laying out your controls by hand. While it is possible to lay out

Java controls by hand, too, you generally won’t want to, for two main reasons. First,

it is very tedious to manually lay out a large number of components. Second,

sometimes the width and height information is not yet available when you need to

arrange some control, because the native toolkit components haven’t been realized.

This is a chicken-

and-egg situation;

it is pretty confusing to figure out when it is okay to use the size of a given

component to position it relative to another.

Each **Container** object has a layout manager associated with it. A layout manager

is an instance of any class that implements the **LayoutManager** interface. The

layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made,

then the default layout manager is used. Whenever a container is resized (or sized

for the first time), the layout manager is used to position each of the components

within it.

The **setLayout( )** method has the following general form:

void setLayout(LayoutManager *layoutObj*)

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable

the layout manager and position components manually, pass **null** for *layoutObj.* If

you do this, you will need to determine the shape and position of each component

manually,using the **setBounds( )** method defined by **Component**. Normally, you will

want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their

names. The layout manager is notified each time you add a component to a

container.

Whenever the container needs to be resized, the layout manager is consulted via its

**minimumLayoutSize( )** and **preferredLayoutSize( )** methods. Each component

that is being managed by a layout manager contains the **getPreferredSize( )** and

**getMinimumSize( )** methods. These return the preferred and minimum size required

to display each component. The layout manager will honor these requests if at all

possible, while maintaining the integrity of the layout policy. You may override

these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined **LayoutManager** classes, several of which are

described next. You can use the layout manager that best fits your application.

***FlowLayout***

**FlowLayout** is the default layout manager. This is the layout manager that the

precedingexamples have used. **FlowLayout** implements a simple layout style, which

is similar to how words flow in a text editor. Components are laid out from the upper-

left corner, left to right and top to bottom. When no more components fit on a line,

the next one appears on the next line. A small space is left between each

component, above and below, as well as left and right. Here are the constructors for

**FlowLayout**:

FlowLayout( )

FlowLayout(int *how*)

FlowLayout(int *how*, int *horz*, int *vert*)

The first form creates the default layout, which centers components and leaves five

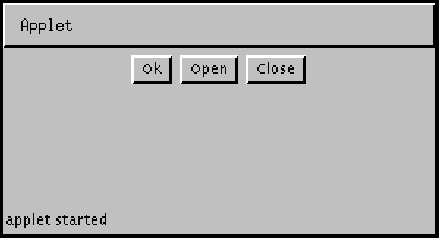
pixels of space between each component. The second form lets you specify how

each line is aligned. Valid values for *how* are as follows:

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT



**Flow layout**

These values specify left, center, and right alignment, respectively. The third form

allows you to specify the horizontal and vertical space left between components in

*horz* and *vert,* respectively.

Here is a version of the **CheckboxDemo** applet shown earlier in this chapter,

modified so that it uses left-aligned flow layout.

// Use left-aligned flow layout.

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*

<applet code="FlowLayoutDemo" width=250 height=200>

</applet>

\*/

public class FlowLayoutDemo extends Applet

implements ItemListener {

String msg = "";

Checkbox Win98, winNT, solaris, mac;

public void init() {

// set left-aligned flow layout

setLayout(new FlowLayout(FlowLayout.LEFT));

Win98 = new Checkbox("Windows 98/XP", null, true);

winNT = new Checkbox("Windows NT/2000");

solaris = new Checkbox("Solaris");

mac = new Checkbox("MacOS");

add(Win98);

add(winNT);

add(solaris);

add(mac);

// register to receive item events

Win98.addItemListener(this);

winNT.addItemListener(this);

solaris.addItemListener(this);

mac.addItemListener(this);

}

// Repaint when status of a check box changes.

public void itemStateChanged(ItemEvent ie) {

repaint();

}// Display current state of the check boxes.

public void paint(Graphics g) {

msg = "Current state: ";

g.drawString(msg, 6, 80);

msg = " Windows 98/XP: " + Win98.getState();

g.drawString(msg, 6, 100);

msg = " Windows NT/2000: " + winNT.getState();

g.drawString(msg, 6, 120);

msg = " Solaris: " + solaris.getState();

g.drawString(msg, 6, 140);

msg = " Mac: " + mac.getState();

g.drawString(msg, 6, 160);

}

}

Following is sample output generated by the **FlowLayoutDemo** applet:

Compare this with the output from the **CheckboxDemo** .

***BorderLayout***

The **BorderLayout** class implements a common layout style for top-level windows. It

has four narrow, fixed-width components at the edges and one large area in the

center.

The four sides are referred to as north, south, east, and west. The middle area is

called the center. Here are the constructors defined by **BorderLayout**:

BorderLayout( )

BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the

horizontal and vertical space left between components in *horz* and *vert,* respectively.

**BorderLayout** defines the following constants that specify the regions:

BorderLayout.CENTER BorderLayout.SOUTH

BorderLayout.EAST BorderLayout.WEST

BorderLayout.NORTH

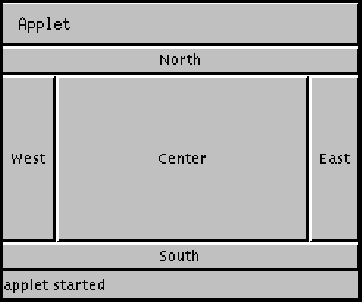
When adding components, you will use these constants with the following form of

**add( )**, which is defined by **Container**:

void add(Component *compObj,* Object *region*);

Here, *compObj* is the component to be added, and *region* specifies where the

component will be added.



**Border layout**

Here is an example of a **BorderLayout** with a component in each layout area:

// Demonstrate BorderLayout.

import java.awt.\*;

import java.applet.\*;

import java.util.\*;

/\*

<applet code="BorderLayoutDemo" width=400 height=200>

</applet>

\*/

public class BorderLayoutDemo extends Applet {

public void init() {

setLayout(new BorderLayout());

add(new Button("This is across the top."),

BorderLayout.NORTH);

add(new Label("The footer message might go here."),

BorderLayout.SOUTH);

add(new Button("Right"), BorderLayout.EAST);

add(new Button("Left"), BorderLayout.WEST);

String msg = "The reasonable man adapts " +

"himself to the world;\n" +

"the unreasonable one persists in " +

"trying to adapt the world to himself.\n"

"Therefore all progress depends " +

"on the unreasonable man.\n\n" +

" - George Bernard Shaw\n\n";

add(new TextArea(msg), BorderLayout.CENTER);

}

}

***GridLayout***

**GridLayout** lays out components in a two-dimensional grid. When you instantiate

a **GridLayout**, you define the number of rows and columns. The constructors

supported by **GridLayout** are shown here:

GridLayout( )

GridLayout(int *numRows*, int *numColumns* )

GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single-column grid layout. The second form creates a grid

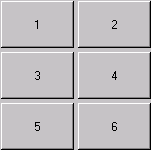
layout with the specified number of rows and columns. The third form allows you

to specify the horizontal and vertical space left between components in *horz* and

*vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying

*numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as

zero allows for unlimited-length rows.



**Grid layout**

Here is a sample program that creates a 44 grid and fills it in with 15 buttons, each

labeled with its index:

// Demonstrate GridLayout

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="GridLayoutDemo" width=300 height=200>

</applet>

\*/

public class GridLayoutDemo extends Applet {

static final int n = 4;

public void init() {

setLayout(new GridLayout(n, n));

setFont(new Font("SansSerif", Font.BOLD, 24));

for(int i = 0; i < n; i++) {

for(int j = 0; j < n; j++) {

int k = i \* n + j;

if(k > 0)

add(new Button("" + k));

}

}

}

}

**Project Code**

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

public class calc extends Frame implements ActionListener{

Checkbox inv; // TO find inversen angle(i.e. sin

//inverse, cos inverse

Checkbox cb[]=new Checkbox[2]; //Radio buttons (to Select

between Degree or Radian)

CheckboxGroup cbg; //checkbox group for above radio

buttons

Button sim[] = new Button[42]; //Actual calculator

buttons

Label mst=new Label(""); //Label to check the status of

memory variable

Boolean btnflag=false; /\*flag to check whether the

last button pressed was a

number(0-9,.) or an operation (+,-,\*...etc.)\*/

double angle; /\*variable used for calculations

of sin,cos,tan and their inverse\*/

String bfrdt="",mem="0"; /\*bfrdt-save first operand of

binary operation

mem- save the memory variable \*/

Label res = new Label("0",Label.RIGHT); //Display of the

calculator.

int j=0; //stores code for last binary

operator.

String r="0"; //used to store the result of

the calculation

int ky; //variable to get key code for

keyboard operations.

public calc() { //constructor to initialize

variables.

Panel main = new Panel(); //Main Panel to hold

overall elements of the calculator.

main.setLayout(new FlowLayout()); //setting the

layout(arrangements of elements)

res.setBounds(0,15,355,30); /\*setting the

boundries of the Display.

left(x)=0 top(y)=15

width=355 height=30\*/

res.setFont(new Font("Helvetica", Font.PLAIN, 20));

/\*setting the font used

in the Display.\*/

res.setForeground(Color.black); //setting the

foreground color(color of text)

res.setBackground(Color.white); //setting the

background color(color behind text)

Panel cal = new Panel(); //panel to hold display and

memory status label

cal.setLayout(new BorderLayout(0,15)); //setting

the layout.

cal.add("North", res); /\*adding diapay label into

the panel 'cal' at top

location\*/

//main.

add("North",cal); //adding cal panel into main panel at

center location

cal.add("North",mst); /\*adding memory status

label into the panel 'cal' at top

location\*/

Panel rkpan = new Panel(); //panel to store

calculator button.

rkpan.setLayout(new GridLayout(10,5,5,5));

/\*setting layout to gridlayout

for Grid view\*/

add("South",rkpan);

for(int i = 0; i < 10; i++)

sim[i]=new Button(""+i);

inv = new Checkbox("Inverse");

cbg= new CheckboxGroup();

cb[0]=new Checkbox("Degrees",cbg,true);

cb[1]=new Checkbox("Radians",cbg,false);

sim[10]=new Button("Back");

sim[11]=new Button("Clear");

sim[12]=new Button("MC");

sim[13]=new Button("MR");

sim[14]=new Button("MS");

sim[15]=new Button("M+");

sim[16]=new Button("sin");

sim[17]=new Button("cos");

sim[18]=new Button("tan");

sim[19]=new Button("x^3");

sim[20]=new Button("x^2");

sim[21]=new Button("+/-");

sim[22]=new Button("sqrt");

sim[23]=new Button("%");

sim[24]=new Button("1/x");

sim[25]=new Button(".");

sim[26]=new Button("Exp");

sim[27]=new Button("log");

sim[28]=new Button("n!");

sim[29]=new Button("pi");

sim[30]=new Button("Not");

sim[31]=new Button("+");

sim[32]=new Button("-");

sim[33]=new Button("\*");

sim[34]=new Button("/");

sim[35]=new Button("x^y");

sim[36]=new Button("Mod");

sim[37]=new Button("And");

sim[38]=new Button("Or");

sim[39]=new Button("Xor");

sim[40]=new Button("=");

sim[41]=new Button("Int");

for(int i = 0; i < 42; i++) {

rkpan.add(sim[i]);

sim[i].addActionListener(this);

sim[i].addKeyListener(new KeyAdapter(){

public void keyReleased(KeyEvent ke){

ky=(int)ke.getKeyChar();

for(int i=48,j=0;i<58;i++,j++)

if(ky==i){

calcu(j);

return;

}

switch(ky){

case 43:calcu(31);

break;

case 45:calcu(32);

break;

case 42:calcu(33);

break;

case 47:calcu(34);

break;

case 38:calcu(37);

break;

case 33:calcu(28);

break;

case 37:calcu(23);

break;

case 61:

case 10:

calcu(40);

break;

case 126:calcu(30);

break;

case 94:calcu(35);

break;

case 8:calcu(10);

break;

case 27:calcu(11);

break;

case 46:calcu(25);

break;

}

}

});

}

for(int i = 0; i < 2; i++)

rkpan.add(cb[i]);

rkpan.add(inv);

//cal.add("South",rkpan);

//setLayout(new BorderLayout(0, 0));

//add("North", main);

addWindowListener(new WindowAdapter(){

public void windowClosing(WindowEvent we) {

System.exit(0);

}

});

}

public void actionPerformed(ActionEvent ae) {

int i;

for(i=0;i<42;i++)

if(ae.getSource() == sim[i])

break;

calcu(i);

}

public void calcu(int i){

String txt=res.getText();

try{

switch(i){

case 0:

case 1:

case 2:

case 3:

case 4:

case 5:

case 6:

case 7:

case 8:

case 9:

if((txt.equals("0"))||(btnflag))

r=""+i;

else

r=txt+i;

btnflag=false;

break;

case 25:if(btnflag)

r="0.";

else if(txt.indexOf('.')==-1)

r=txt+".";

btnflag=false;

break;

default:

btnflag=true;

switch(i){

case 10:if(txt.length()==1)

r="0";

else

r=txt.substring(0,txt.length()-1);

break;

case 11:r="0";

j=0;

break;

case 12:mem="0";

mst.setText("");

break;

case 13:r=""+mem;

break;

case 14:if(txt!="0"){

mem=txt;

mst.setText("M");

}

break;

case 15:if(txt!="0"){

mem=""+(Double.parseDouble(mem)+Double.parseDouble(txt));

mst.setText("M");

}

break;

case 16:if(!inv.getState()){

if(cbg.getSelectedCheckbox()==cb[0])

angle=(Math.PI\*(Double.parseDouble(txt))/180);

else

angle=Double.parseDouble(txt);

r=""+Math.sin(angle);

}

else{

if(cbg.getSelectedCheckbox()==cb[0])

r=""+(Math.asin(Double.parseDouble(txt))\*180/Math.PI);

else

r=""+Math.asin(Double.parseDouble(txt));

inv.setState(false);

}

break;

case 17:if(!inv.getState()){

if(cbg.getSelectedCheckbox()==cb[0])

angle=(Math.PI\*(Double.parseDouble(txt))/180);

else

angle=Double.parseDouble(txt);

r=""+Math.cos(angle);

}

else{

if(cbg.getSelectedCheckbox()==cb[0])

r=""+(Math.acos(Double.parseDouble(txt))\*180/Math.PI);

else

r=""+Math.acos(Double.parseDouble(txt));

inv.setState(false);

}

break;

case 18:if(!inv.getState()){

if(cbg.getSelectedCheckbox()==cb[0])

angle=(Math.PI\*(Double.parseDouble(txt))/180);

else

angle=Double.parseDouble(txt);

r=""+Math.tan(angle);

}

else{

if(cbg.getSelectedCheckbox()==cb[0])

r=""+(Math.atan(Double.parseDouble(txt))\*180/Math.PI);

else

r=""+Math.atan(Double.parseDouble(txt));

inv.setState(false);

}

break;

case 19:r=""+Math.pow((Double.parseDouble(txt)),3);

break;

case 20:r=""+Math.pow((Double.parseDouble(txt)),2);

break;

case 21:r=""+(0-Double.parseDouble(txt));

break;

case 22:r=""+Math.sqrt(Double.parseDouble(txt));

break;

case 23:r=""+Float.parseFloat(txt)/100;

break;

case 24:r=""+1/Float.parseFloat(txt);

break;

case 26:r=""+Math.exp(Double.parseDouble(txt));

break;

case 27:r=""+Math.log(Double.parseDouble(txt));

break;

case 28:r=""+fact(Integer.parseInt(txt));

break;

case 29:r=""+Math.PI;

break;

case 30:r=""+(~Integer.parseInt(txt));

break;

case 41:r=""+Math.rint(Double.parseDouble(txt));

break;

case 31:

case 32:

case 33:

case 34:

case 35:

case 36:

case 37:

case 38:

case 39:

case 40:

switch(j){

case31:r=""+(Double.parseDouble(txt)+Double.parseDouble(b

frdt));

break;

case 32:r=""+(Double.parseDouble(bfrdt)-

Double.parseDouble(txt));

break;

case33:r=""+(Double.parseDouble(bfrdt)\*Double.parseDouble(txt));

break;

case34:r=""+(Double.parseDouble(bfrdt)/Double.parseDouble(txt)

);

break;

case35:r=""+Math.pow(Double.parseDouble(bfrdt),Double.par

seDouble(txt));

break;

case36:r=""+(Double.parseDouble(bfrdt)%Double.parseDouble

(txt));

break;

case37:r=""+(Integer.parseInt(bfrdt)&Integer.parseInt(txt

));

break;

case38:r=""+(Integer.parseInt(bfrdt)|Integer.parseInt(txt

));

break;

case39:r=""+(Integer.parseInt(bfrdt)^Integer.parseInt(txt

));

break;

}

j=i;

bfrdt=r;

break;

}

}

if(r.equals("NaN"))

r="Invalid input provided for the function";

int l=r.length();

if((l>1)&&(r.charAt(l-1)=='0')&&(r.charAt(l-2)=='.'))

r=r.substring(0,l-2);

res.setText(r);

}catch(Exception e){

bfrdt="0";

j=0;

res.setText("Invalid input provided for the function");

}

}

double fact(int n)

{

if(n==0)

return(1);

else

return(n\*fact(n-1));

}

public static void main(String args[]) {

calc cal = new calc();

cal.setSize(370,350);

cal.setTitle("Calculator By KAILASH PRAKASH PAVAN");

cal.setFont(new Font("SansSerif", Font.BOLD, 12));

cal.setBackground(Color.gray);

cal.setVisible(true);

cal.setResizable(false);

}

}

**TESTING**

**Software testing techniques:**

Testing provides an interesting anomaly for the software engineers during earlier software engineering activities. The engineer’s attempts to build software from abstract software to a tangible product. Now comes testing. The engineer create a series of test that are intended to “demolish” the software has been built. Infect testing is one step in the software process that could be viewed as distinctive rather than constructive. Software engineer are by their nature constructive people. Testing requires that the developer discard preconceived. Notions of the “correctness” of software just developed and overcome a conflict of interest that occurs when error are uncovered.

**Testing objectives:**

Software testing states a number of rules that can serve as testing objectives.

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as yet undiscovered error.
3. A successful test is one that uncovers as a yet undiscovered error.

**Testing principles:**

Before applying method to design effective test cases we must understand the basic principle that guides the software testing.

**All tests should be traceable to customer requirements.** As we know the objective of software testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.

**Test should be planed long before testing begins.**

Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore all tests can be planned and designed before any code has been generated. The most effective testing should be conducted by an independent third party. By most effective we may test that has the highest probability of finding errors.

**Black box testing:**

Black box testing was also called Behavioral testing focus on the functional requirements of the software that is black box testing enables the software engineer to derive set of inputs condition and that will fully exercise all functional requirement of the program. Black box testing is not an alternating to white box testing rather it is complimentary approach that is likely to uncover a different classes of errors that white box method. Black box testing attempts to find errors in the following categories.

(1) Incorrect or missing function

(2) Interface errors

(3) Error in data structure or external data base access.

(4) Behavior or performance errors

(5) Initializations or termination errors

**Testing GUIs:**

Graphical user interface (GUI) present interesting challenges for software engineers because of reusable components provided as a part of GUI development environments. The creation of user interface has become less time consuming and more precise but at the same time the complexity of GUI has grown leading to more difficulty in the design and execution of test cases because many modern GUIs have the same look and feel a series of standard test can be derived finites states modeling graphs may be used to derive a series of test that address specific data and program objects that are relevant to GUI.

**TEST CASES:**

The primary objective for test case design is to derive a set of tests that have the highest likelihood for uncovering errors in the software.

Following is the table of test cases along with there results performed during the test phase of the project development.

**1. SHOW WHOLE LIST module:**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Test case** | **Expected Result** | **Actual Result** |
| **1.** | **Check if digits is displayed in proper way** | **Should display** | **As expected** |

**2,SHOW INDIVIDUAL module:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test case** | **Expected Result** | **Actual Result** |
| **1.** | **Check if the total no of digits are displayed** | **Should display** | **As expected** |
| **2.** | **Check if user entries are displayed in proper way** | **Should display** | **As expected** |

1. **MODIFY ENTRIES Module**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Test Cases** | **Expected Result** | **Actual Result** |
| **1.** | **If the user has entered digits without limit** | **Should display error** | **As expected** |
| **2.** | **Check weather user is allowed to reenter the digits** | **Should allow** | **As expected** |

Security

The key that allows java to solve the problem of security just described is that the output of

java compiler is not executable code.Rather ,it is bytecode .Bytecode is a highly optimized

set of instructions designed to be executed by java run time system,which is called java

virtual machine(JVM).

Translating a java program into bytecode makes it much easier to run a program

in a wide variety of environments because only the JVM needs to be implemented for each

platform.Once the run-time package exists for a given system ,any java program can run on

it.Thus,the execution of bytecode by the JVM is easiest way to create truly portable

programs.

The fact that a java program is executed by the JVM also helps to make it

secure.Because the JVM is in control,it can contain the program and prevent it from

generating side effect outside of the system.

|  |
| --- |
| 1. [1 Calculator: frequently asked questions](mshelp://Windows/?id=f15f7d3e-ee9c-465a-a7e8-4e6af5cfee5d) |
| 1. [2 Open Calculator](mshelp://Windows/?id=309f381e-6c92-4ad1-b685-f4c2d7ceb3a5) |
| 1. [3 About certificate errors](mshelp://Windows/?id=064af181-10b3-49e7-9eb6-ba3f07d88803) |
| 1. [4 Get information about Secure Sockets Layer (SSL) certificates](mshelp://Windows/?id=ddfa63ef-15bd-44fd-82e3-cb6eb5f5252a) |
| 1. [5 How to know if an online transaction is secure](mshelp://Windows/?id=16ee1fa9-60ab-4c1b-9b35-15b59a142f43) |
|  |
| 1. [6 Using Windows Security Center](mshelp://Windows/?id=c5e78ee2-b00a-444d-8c57-e29bda8768a8) |
| 1. [7 Windows security information for IT Pros](mshelp://Windows/?id=8a3f4523-7237-41da-aad5-75309a9a0904) |
| 1. [8 Which settings affect security in Windows Media Player?](mshelp://Windows/?id=0f840d98-44f1-42d6-9553-30ef37306229) |
| 1. 9 Understanding security and safe computing |
| 1. [10 Privacy and security considerations when sharing media in Windows Media Player](mshelp://Windows/?id=c6e156f0-1871-409c-8b05-3e8bd7db3976) |
| 1. [11 Changing intranet security settings](mshelp://Windows/?id=9cf3477e-2a6e-4c18-bf42-4ce760e952a9) |
| 1. [12 Get security updates for Windows](mshelp://Windows/?id=8ee4f673-5b5c-4de7-a2df-ed0b3d2d5469) |
| 1. [13 Demo: Security basics](mshelp://Windows/?id=8508a3e5-0c06-411f-b5a6-383f84d19c25) |
| 1. 14 Security and privacy features in Internet Explore |
| 1. [15 Setting up web server security](mshelp://Windows/?id=7ac7e97d-1094-4041-9012-f555eae186df) |
| 1. [16 Change Internet Explorer Security settings](mshelp://Windows/?id=c9a5706f-0596-424f-bdfa-59618cb136e2) |
| 1. [17 Making your network more secure](mshelp://Windows/?id=bf6372bb-c95c-4b39-aa50-de7e5bf03468) |
| 1. [18 About Tablet PC Input Panel](mshelp://Windows/?id=946d3cc2-943d-4b85-8c3c-6e741b4f8234) |
| 1. [19 Windows Security Guide for IT Pros](mshelp://Windows/?id=b93bb9c9-51e1-4484-9a38-ec76a6b17d77) |
| 1. [20 Security zones: adding or removing websites](mshelp://Windows/?id=fd277a6b-3722-445b-b32e-1f8e925c385a) |
| 1. [21 Web browsing and security: recommended links](mshelp://Windows/?id=22ee7c6c-41ff-4311-bedc-0b29f4a0e6b8) |
| 1. [22 Password security with Tablet PC Input Panel](mshelp://Windows/?id=60b293ab-7242-4d41-a4d6-eb6c9383ca13) |
| 1. [23 What are the different wireless network security methods?](mshelp://Windows/?id=b385cc8a-af25-489e-a82e-decf6df26b68) |
| 1. [24 Set the Windows Mail security level](mshelp://Windows/?id=7676cc8b-2e7a-4280-a862-3cc177235023) |
| 1. [25 Using Security Center at work](mshelp://Windows/?id=af7df38c-4392-4144-ab7b-9476a01d678d) |
| 1. [26 Mobile PC security basics](mshelp://Windows/?id=dfd5fdd6-053f-42c1-af70-bf9403e6d01a) |
| 1. [27 How do I know if a wireless network is secure?](mshelp://Windows/?id=80ec1c9e-4d3d-4008-a341-ebe1474b247f) |
| 1. [28 Advanced information about Internet Explorer](mshelp://Windows/?id=44be8b8a-78dd-4bb7-b7f5-1f0e40b3c8b5) |
| 1. [29 Change Windows Mail security settings](mshelp://Windows/?id=de968ece-5571-4504-9b3f-1d6475c43331) |
| 1. [30 Security and privacy in Windows Mail](mshelp://Windows/?id=e39eebdb-b9d6-4a30-80dd-43edf496acb1) |

**LIMITATION OF THE PROJECT**

|  |
| --- |
| * [1 Calculator: frequently asked questions](mshelp://Windows/?id=f15f7d3e-ee9c-465a-a7e8-4e6af5cfee5d) |
| * [2 Open Calculator](mshelp://Windows/?id=309f381e-6c92-4ad1-b685-f4c2d7ceb3a5) |
| * [3 Send and Receive PIM items](mshelp://OEM/?id=brcm_send_receive_pim) |
| * [4 Introduction to computers](mshelp://Windows/?id=ef0ebe45-7b6d-42bf-a40d-6e83359c1144) |
| * [5 Configure Bluetooth sharing](mshelp://OEM/?id=brcm_send_receive_file) |
| * [6 Use and configure your Bluetooth enabled device (unknown device type)](mshelp://OEM/?id=brcm_unknown_ops_settings) |
| * [7 Burn a CD or DVD in Windows Media Player](mshelp://Windows/?id=30ffa641-b682-4315-8c7e-8a29f383a713) |
| * [8 Use and configure your Bluetooth enabled headset or headphones](mshelp://OEM/?id=brcm_headset_ops_settings) |
| * [9 Limit the content that children can view on the web](mshelp://Windows/?id=ff9aa367-0420-4b84-86cd-45bd898fccef) |
| * [10 Use and configure a remote Bluetooth enabled computer](mshelp://OEM/?id=brcm_computer_ops_settings) |

**FUTURE SCOPE AND APPLICATION**

|  |
| --- |
| * [1 Developing applications for Internet Information Services (IIS) 7.0](mshelp://Windows/?id=a8075305-2a45-4e84-bbcc-7e73c87db82d) |
| * [2 Application management strategies for IT Pros](mshelp://Windows/?id=fe6f2c27-067a-4965-a786-2d486ebffd16) |
| * 3 |
| * [4 How do I mark an application to always require a full administrator access token (User Account Control)?](mshelp://Windows/?id=12ad62e6-f51b-4eaa-8352-8513589c59e7) |
| * [5 Publishing applications with Internet Information Services (IIS) 7.0](mshelp://Windows/?id=8faabd36-9c41-4e24-9934-71752edac768) |
| * [6 How do I enable and disable application setup detection in User Account Control?](mshelp://Windows/?id=94b71b51-e2ef-4178-b984-02110cdc792e) |
| * [7 Use and configure your Bluetooth enabled headset or headphones](mshelp://OEM/?id=brcm_headset_ops_settings) |
| * [8 Send and Receive PIM items](mshelp://OEM/?id=brcm_send_receive_pim) |
| * [9 Use and configure your Bluetooth enabled device (unknown device type)](mshelp://OEM/?id=brcm_unknown_ops_settings) |
| * [10 Use and configure your Bluetooth enabled PDA](mshelp://OEM/?id=brcm_pda_ops_settings) |
| * [11 Use and configure your Bluetooth enabled phone](mshelp://OEM/?id=brcm_phone_ops_settings) |
| * [12 Use and configure a remote Bluetooth enabled computer](mshelp://OEM/?id=brcm_computer_ops_settings) |
| * [13 Use and configure your Bluetooth enabled camera](mshelp://OEM/?id=brcm_camera_ops_settings) |
| * [14 Use and configure your Bluetooth enabled modem](mshelp://OEM/?id=brcm_modem_ops_settings) |
| * [15 Monitoring Internet Information Services (IIS) servers, sites, and applications](mshelp://Windows/?id=b768fbac-b430-46d5-9b57-202a36f39d99) |
| * [16 Setting up support for applications](mshelp://Windows/?id=509b95c1-ed06-47ba-9f8b-2feb88945038) |
| * [17 Program (definition)](mshelp://Windows/?id=9ada1558-de2c-46fa-b6f3-5330af66ac9b) |
| * [18 Configure Bluetooth sharing](mshelp://OEM/?id=brcm_send_receive_file) |

**BIBLIOGRAPHY**

1. Herbert Schildt(TMH) , The Complete Reference java 7

2. E.Balaguruswami(TMH),Java 2.0 Programming

3. R.Krishnamurthy & Prabhu,Internet & java Programming